FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Velocity estimation for Autonomous Underwater Vehicles using vision-based systems

Hélio Manuel Silva Puga

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Supervisor: José Carlos dos Santos Alves

Co-supervisor: Andry Maykol Pinto

June 25, 2018

Resumo

Uma variável importante para a navegação de submarinos autónomos (sigla em Inglês: AUV) é a sua velocidade, no entanto, estimar a velocidade não é uma tarefa fácil de realizar. Métodos tradicionais, como o Sistema de Posicionamento Global (GPS) não funcionam debaixo de água, e os outros métodos envolvem calcular a velocidade relativamente a correntes marítimas ou usar sensores acústicos. Porém, estes sistemas têm as suas limitações. Uma solução para calcular a velocidade do AUV é usar sistemas baseados em visão computacional quando uma estrutura marítima está no campo de visão do AUV. Este tipo de sistemas têm a vantagem de não estarem sujeitos aos mesmos problemas que os sistemas de navegação acústica, bem como ao efeito das correntes marítimas.

Neste trabalho é apresentado um estudo de um sistema capaz de calcular, em tempo real, a velocidade de translação e angular de um AUV, possível de ser usada no controlo do veículo. Para estimar o movimento entre *frames*, através do uso de uma camera monocular, foram estudados diferentes algoritmos de visão computacional capazes de estimar o campo de movimento da *frame*, isto é, o movimento de uma região ou de todos os píxeis da frame, num stream de vídeo. Os algoritmos estudados foram o Lucas e Kanade e a sua versão iterativa e em pirâmide, e correspondência de blocos. Os algoritmos foram estudados e adaptados para corresponderem aos requisitos de navegação dos AUVs, e às características do movimento destes veículos, como a seu baixo valor de aceleração e máxima velocidade. Também são discutidos os problemas inerentes a sistemas de visão computacional, como o *aperture problem* e o *parallax effect*. As limitações destes algoritmos também foram apresentadas como o maxima deslocação entre duas frames consecutivas.

Também foi estudado e desenvolvido um algoritmo capaz de estimar a velocidade angular e de translação do veículo através do campo de movimento. Os algoritmos foram testados em cenários simulados e foram apresentados os resultados provenientes do uso dos diferentes métodos para estimação do campo de movimento visual. Foi concluído que o algoritmo de estimação de velociodade funciona como pretendido e deverá ser usado em conjunto com a versão iterativa e em pirâmide do Lucas e Kanade, que foi a que produziu os melhores resultados, com um erro máximo de 3% para os testes realizados, e mostrou ser mais robusta em condições difíceis.

Considerando que os algoritmos de visão computacional, para estimação do campo de movimento, são computacionalmente exigentes logo, incompatíveis com um sistema de tempo real quando implementados em microcomputadores, este problema foi contornado estudando uma possível implementação da versão iterativa e em pirâmide do Lucas e Kanade, numa Field Programable Gate Array (FPGA) e em um microcomputador. Cada etapa do algoritmo para estimação do campo de movimento está detalhada assim como os seus parâmetros. Para o algoritmo de estimação da velocidade angular e de translação também foi detalhada e explicada cada etapa. Por fim, foi proposta uma arquitetura, possível de ser implementada numa FPGA, assim como uma estimativa do consumo de recursos. Este trabalho funciona como um ponto de partida para a implementação de um sensor para estimação de velocidade num sistema embebido.

Abstract

One important variable for Autonomous Underwater Vehicles (AUV) navigation is their velocity; however, estimating it is not an easy task. Traditional methods, like Global Position System (GPS), does not work underwater, and the other methods involve computing the velocity relative to the sea currents or using acoustic sensors. Nevertheless, these systems also have their limitations. One solution to compute the AUVs velocity is using computer vision based systems when a maritime structure is in the AUVs field of view, as this type of sensors are not subjected to the same problems as the acoustic navigation as well as the effect of sea currents.

In this work, it is presented a study of a system capable of calculating the linear and angular velocity of an AUV, in real time, suitable to be used in the control loop of the vehicle. To estimate the motion, using monocular camera, it were studied three different computer vision algorithms capable of estimating the motion field of the video stream. The algorithms studied were the Lucas and Kanade, and its iterative and pyramidal implementation, and block matching. These algorithms were study and adapted to fit the requirements for AUV navigation and, keeping in mind the movement characteristics of this vehicles, such as slow acceleration and maximum velocity. It were also discussed the inherent problems to the vision based systems, such as the aperture problem and the parallax effect. The limitations of the algorithms are also presented, such as the maximum displacement between frames.

It was also studied and developed an algorithm capable of estimating the vehicle angular and translational velocity using the motion field computed from the visual motion estimation algorithm. The algorithms were tested in simulated scenarios and it was presented the results using the different methods for motion estimation. It was concluded that the velocity estimation algorithm preforms as expected and should be used with the iterative and pyramidal Lucas and Kanade, as it yield better results, with a maximum error of 3% for the performed tests, and showed to be more robust in difficult conditions.

Considering that the computer vision algorithms, for motion estimation, are computing intensive thus not compatible with real-time system when implemented in microcomputers, this problem was solved though the study of a possible implementation of the iterative and pyramidal Lucas and Kanade in a Field Programmable Gate Array (FPGA), and in a microcomputer.

Every stage of the algorithm to compute the motion estimation is detailed as well its parameters. Also the algorithm for velocity estimation is explained and detailed. An hardware architecture and an estimation of the resource consumption is presented. This work lays down a starting point to the implementation of a velocity estimation sensor an embedded system with an FPGA.

Acknowledges

I would like to thank to my advisor doctor José Carlos Alves from Faculty of Engineering of University of Port, for all its availability to answer my questions and for all the constructive feedback throughout this work. I would also like to acknowledge doctor Andry Maykol Pinto for all your help and feedback.

Thank you to all my friends for the good relaxing moments and enjoyment during this dissertation, as well as the availability to help and support when needed.

Thank you to my family for all the affection and support demonstrated, not just during this dissertation but, as well during all the last 5 years.

Hélio Puga

"The way to succeed is to double your failure rate."

-Thomas J. Watson

Contents

List of Figures

Abbreviations and Symbols

2D	Two dimensions
3D	Tree dimensions
ASIC	Application-specific integrated circuit
AUV	Autonomous Underwater Vehicle
DR	Dead-reckon
DVL	Doppler Velocity Logs
FIFO	First in first out
FPGA	Field-programmable gate array
FPS	Frames per second
GPS	Global Positioning System
LUT	Lookup table

Chapter 1

Introduction

1.1 Context

The ocean plays an important role in the Earth's biosphere and economy as it is home to a large number of species and one of the largest sources of resources on Earth, such as oil and natural gas [3]. Due to its vastness, the majority of the ocean is still unexplored, and its harsh environment places a barrier to exploration, maintenance, and supervision of underwater structures, such as power transmission cables and sub-sea pipeline, and ecosystems [4]. Therefore, there is an increasing interest in developing technology that performs such tasks in a practical and safe way.

As the economy grows, there is a need for more resources. Moreover, the emerging of human concern about the well-being of underwater ecosystems and the will to unveil the oceans unknowns with scientific explorations make underwater activities to be on the rise. These activities are usually done with teams of specialized divers, but due to ocean's environment and biodiversity there are some risks of human losses. Furthermore, in some cases it is impossible for humans to perform the task manually, for exemple the inspection of man made structures on the sea floor, as a submarine cables or pipelines, or the exploration of the sea bottom for searching valuable materials. It is in these cases that the use of Autonomous Underwater Vehicle (AUV), a vehicle that travels underwater without user inputs, becomes essential. These vehicles allow navigation to high depths and in hazardous environments with no risk for humans [4].

The AUV's velocity is one of the most important variables during navigation. Although estimating the velocity relative to water is easy and can be done with different sensors, this measurement is unreliable for obtaining the velocity relative to the ocean floor or even to a fixed or moving submerged structures because it ignores the effect of sea currents, making it unreliable for obtaining the exact location of an AUV [5].

In situations where good precision is of utmost importance, for example, when the AUV is close to maritime structures, one viable solution is using vision-based systems to estimate orientation and velocity relative to structures of interest [5]. One way of obtaining the velocity is using optical flow which consists in calculating the change of two, or more, consecutive images and from that calculate the velocity.

1.2 Motivation

Obtaining the velocity vector has requirements of a real-time system, since a velocity vector that exceeds its time validity is useless in a control loop. However, calculating the optical flow and extracting the vehicle velocity is a computing intensive task and might not be compatible with the limited processing speed of low power embedded computing systems, usually onboard of small AUVs. Therefore, it becomes very difficult to obtain a velocity vector in useful time [6].

To solve the problem described above, in this dissertation we aim to develop a system capable of estimating the velocity of an AUV relative to a structure seen by the camera, namely the sea floor, using optical flow techniques. To overcome the limitations due to high computational complexity of the known optical flow algorithms, a low resolution image is considered. The real-time computation of the dense optical flow is purposed to be done by a custom computing system implemented in a FPGA-based system. This pre-processing will accelerate the task of optical flow calculation by: a) performing some operations in the image resulting in a faster optical flow calculation in the embedded computer, or b) dividing the calculus by the embedded computer and an FPGA. The resulting data will be used to obtain the vehicle velocity and angular velocity.

1.3 Goals

This dissertation's goal is to develop a system to estimate the velocity and the angular velocity for a real-time application in an AUV attending to computational power limitations. Our main objectives are:

1. Research and study of Optical flow algorithms;
2. Evaluation of the performance of optical flow algorithms and definition of their parameters;
3. Study and evaluation of an algorithm for velocity estimation;
4. Implementation and characterisation of the final solution.

1.4 Contributions

The main contributions of this work are:

1. Description of each stage of the optical flow computation and implementation, namely the basic version of the Lucas and Kanade and its iterative and pyramidal approach, as well as block matching algorithm. These algorithms are adapted for computing vehicles velocity and implementation in embedded systems. Moreover, it is performed an analysis and a comparison between the different methods.
2. Description of the algorithm to compute the translational and angular velocity from the motion field, obtained form the optical flow algorithms. It was, also, performed tests and an evaluation of the algorithm.

3. Architecture proposal and discussion for implementation of the iterative and pyramidal Lucas and Kanade in a FPGA.

1.5 Organization of this document

The document is organized as follows:

Chapter 2 Presents a bibliographic review of the state of the art solutions for velocity measurements, navigation and localization of the AUVs, with main focus on optical geophysical navigation. It is presented the main optical flow methods, capable of being implemented in embedded systems, namely Lucas and Kanade and Horn and Schunck. Also, it is presented state of the art implementations of these algorithms in FPGA.

Chapter 3 In this chapter are detailed the steps of Lucas and Kanade basic algorithm and its iterative and pyramidal approach. It is also presented the method block matching, which consists in an alternative method for estimation of image movement. Even more, is presented the results and discussion for each one of the methods.

Chapter 4 The algorithm for estimating the velocity from the motion field is presented at this chapter. It is detailed the process to estimate the translational velocity and the angular velocity. The results of the method, using the different methods to estimate the image movement in the chapter 4, are presented and discussed.

Chapter 5 An FPGA architecture is presented and a estimative of the resource consumption and performance.

Chapter 6 It is presented the main conclusions of this work, and it is purposed some future work.

Chapter 2

Bibliographic review

In this chapter, we will analyse the existing state of the art solutions for velocity measurement, navigation, and localization of an AUV (section 2.1). Then we will focus in the optical systems that are used for navigation (section 2.2), afterwards, it is introduced the concept of optical flow and the different existing algorithm for optical flow computation (section 2.3). Furthermore, we will give an overview of some works implementing optical flow in hardware from the last years (section 2.4). At last, will be presented how the 3D motion of the camera creates a 2D motion field (section 2.5).

2.1 Navigation in AUV

The navigation and localization of an AUV is a challenging task. Above water, most navigational systems rely on radio or spread-spectrum communication and GPS. However, when underwater, due to strong electromagnetic radiation absorption of the ocean, electromagnetic communication is unavailable to AUVs. This means that traditional RF-based systems cannot be used for AUV navigation [1, 5]. In order to solve this problem, as described in [1], nowadays we rely on three main techniques: (i) inertial/dead-reckoning; (ii) acoustic transponders and modems; (iii) geophysical. An overview of the AUV navigation and localization techniques is presented in the figure 2.1.

2.1.1 Inertial/Dead-Reckoning

This type of navigation is achieved when the AUV positions himself without support from acoustic transponders, ship or GPS. In dead-reckoning, (DR) the AUV controls himself based on the velocity and acceleration vectors obtained from accelerometers, gyroscopes, water speed sensors, and Doppler velocity loggers (DVL) [1, 5]. Keeping in mind that ocean currents can go up to 3 km/h [7] and AUV generally operate at low velocities, the main problem with DR is that the presence of an ocean current will add an undetected velocity component to the vehicle.

In order to try to minimize the effects of the sea currents, it was integrated into the inertial navigation system a Doppler velocity logger for operation near the seabed. This type of sensor uses Doppler shift to measure AUV velocity relative to the seabed. In the end, this integrated

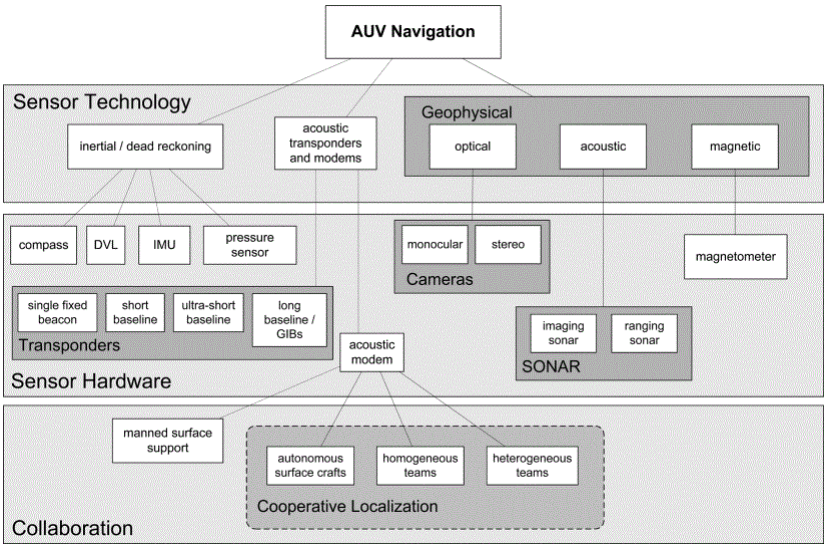


Figure 2.1: Classification of AUV navigation and methods used as described in [1].

solution can achieve drifts from 0.1% for high-end units, to 2%-5% for modestly priced units [1]. Nevertheless, the methods in this category exhibit a position error that is unbounded [1].

2.1.2 Acoustic transponders and modems

In this category, the position of the AUV is based on measuring the time of flight of signals between acoustic beacons or modems, placed at known locations, and the AUV [1]. The basic function principle is: initially the vehicle sends an acoustic signal that is returned for each one of the beacons which have received it. Then, it is measured the time of flight, and knowing the local sound speed, and the configuration of the beacons array, the distance to each beacon is computed and combined with the depth information (from a depth sensor and/or a sonar to measure the distance to the sea bottom) a 3D location solution is obtained. Lastly, it is possible to compute the point of intersection between, at least, three spheres, being this point the AUV location [5].

The techniques in this category are all subject to the accurate measurement of the speed of sound in the water. However, this is a difficult task as the water pressure rises with ocean depth, and temperature variations make the speed of sound change. In addition, acoustic multipath will result in an incorrect time of flight calculation and hence erroneous position fixes [1, 5]. Furthermore, these systems require the setup to support the additional hardware infrastructure, as they need beacons that must be set up before navigation can begin, adding costs to the operation.

2.1.3 Geophysical

As defined in [1], the techniques in this category use the external environmental information to determine AUV localization. This is done with sensors that are capable of extracting and identify environmental features. Almost all methods in this category achieve bounded position error by

some form of simultaneous localization and mapping (SLAM). The geophysical approach includes 3 main categories:

- Magnetic: use of magnetic fields maps for localization and navigation.
- Sonar: use of acoustic sensors to detect and identify features in the environment that can be used as landmarks for navigation purposes.
- Optical: use of cameras to capture images of the seabed and then process this images to navigate. In this work, our main focus will be in this category of navigation and this topic will be developed in the next sections.

2.2 Optical geophysical navigation

The process of determining the robot positioning by analyzing consecutive camera images is called visual odometry. The strategies for obtaining visual odometry are optical flow or structure from motion [1]. The algorithms used in this form of navigation usually operate in symbiosis with other systems, such as the inertial and acoustic. This system aids in navigation increasing precision when close to the structure of interest. In the work [8] are used forward plus downward cameras to correct erroneous positions from dead-reckoning navigation.

One of the algorithms used in optical navigation is the optical flow extracted from the onboard cameras of the AUV. In [9], it is presented a docking and control system that uses optical flow for controlling and docking an AUV. Optical flow is also used in AUVs for target tracking and speed estimations [10].

Although optical methods have advantages over acoustic systems, they still have their own problems. AUVs have power restrictions, as they can not transport large batteries for power supply and they need to stay underwater large amounts of time. Moreover, as AUVs also have space restrictions to accommodate large amounts of equipment for diverse purposes, the computing systems for image processing are usually restricted to small and low power, hence performance limited, computing systems [10].

2.3 Optical flow

Optical flow, as defined by Horn in [11], consists in the apparent motion of brightness patterns observed when a camera is moving relatively to the objects being imaged. Therefore, optical flow estimation consists of the computation of the motion field that represents the pixel displacements between successive frames, which can be defined as the vector (u,v), where u is the horizontal and v is the vertical displacement of each pixel. The first approach, in 1981, by Horn and Schunck[12] and by Lucas and Kanade[13], and most of the work that came after that, is based on the brightness constancy assumption:

$$I(x,y,t) = I(x+u, y+v, t+1) \tag{2.1}$$

where $I(x,y,t)$ represents the intensity of the image pixel at position (x,y) and time t and (u,v)represent the displacement of that same pixel along x and y, respectively.

This means that the intensity I of the pixel in the position (x,y) of the image at the time t is the same as the pixel $(x+u,y+v)$ at the time $t+1$. However, this assumption is easily invalidated by changes in the image brightness due to ambient light rather than from the moving object.

Using the Taylor expansion of 2.1 and assuming the intensity is conserved, this is $\frac{d}{dt}I(x,y,t) = 0$, we derive:

$$I_x u + I_y v + I_t = 0 \tag{2.2}$$

where I_x, I_y, and I_t represent the directional intensity gradients in x, y, and t, respectively.

The flow velocity vector has two unknown variables and, as we only have one equation (2.2), it is not possible to estimate the optical flow, and we must add another constraint to the problem. It is at this point that most gradient-based methods diverge, including the Horn-Schunck and Lucas-Kanade methods. Both of them will be analysed further in the next section.

In 1993, a framework for robustly estimating optical flow was presented by Black and Anandan [14]. In this work, it was addressed the possibility of errors, referred as outliers, due to the violation of the brightness constancy and spatial smoothness assumption. As classical methods [12, 13] use least-squares formulations of optical flow that do not cope well with outliers, in this work is presented a robust framework to calculate the optical flow by reducing outliers influence.

By 2006, in [15], it was presented an algorithm to go beyond the accuracy limits of that time current optical flow algorithms. By the time, some of the algorithms employed a coarse-to-fine approach, which consists in estimating the flow in an image pyramid. At the top is the image at a coarse scale, and the levels beneath it are a warped representation of the images based on the flow at the preceding level. This ensures that the small motion assumption remains valid. Higher accuracy can be achieved using a coarse to over-fine approach which is similar to coarse-to-fine but, instead of stopping at the fine level, the image is interpolated to obtain over-fine levels.

Although optical flow accuracy has greatly improved, most of the methods still are very computational demanding. In [16], an HybridTree method improves the computation time and accuracy by using Horn-Schunk and Lucas-Kanade methods. This improvement comes from preselecting regions in the image where these methods work best, and at the same time improving the runtime for the calculation by not doing unnecessary calculations.

In [6], a database and an evaluation methodology for optical flow was developed, proposing a benchmark and evaluation methods for testing different aspects of optical flow algorithms. This database also includes tables[1] with the results for over 100 methods. One important conclusion retrieved from the online table in [17] is that optical flow algorithms have difficulties with real-time computation as most of them are very complicated and computational demanding, and runtimes are generaly in the order of seconds or even minutes to process a single frame. This makes clear that it is necessary to make a tradeoff between the accuracy of optical flow and the efficiency.

[1] Tables URL: `vision.middlebury.edu/flow/`

However, over the last years, a large number of different algorithms and approaches to the optical flow problem have been developed. As a result, a deep analysis of the evolution and algorithms is an extensive and complex task on its own and would extend this thesis beyond its scope, hence we will only present a more detailed version of tree methods: 1) Horn and Schunck [12], 2) Lucas and Kanade [13] and 3) HybridTree method [16].

2.3.1 Horn and Schunck

The Horn-Schunck [12] approach is one of the gradient-based methods, which uses a spatio-temporal derivation of the intensity I of the image to calculate the optical flow. Moreover, it is possible to calculate the dense optical flow, i. e., calculate it for every pixel in the image.

Simplifying and abstracting from the mathematical deduction in the original paper, using the formulation from [18], it is possible to estimate the velocity field $s(p,t) = (u(p,t), v(p,t))$, where p is the pixel position, (x,y), and t the time, minimizing the energy function given by:

$$E = \int\int [(I_x u + I_y v + I_t)^2 + \alpha^2(||\nabla u||_2^2 + ||\nabla v||_2^2)]dxdy \tag{2.3}$$

being I_x, I_y and I_t the image derivatives in x, y, and time, and the α parameter is a term used for controlling the impact of the smoothing factor. The first member of this equation comes from 2.2 and the second is the smoothness constraint, derived from the assumption that neighbouring pixels in the image are likely to belong in the same surface, so the flow must vary smoothly across the velocity field.

Solving the optical flow problem then becomes a problem of minimizing the equation 2.3. The authors proposed an iterative method with two equations:

$$u_{k+1} = \overline{u}_k - \frac{I_x[I_x\overline{u}_k + I_y\overline{v}_k + I_t]}{\alpha^2 + I_x^2 + I_y^2} \tag{2.4}$$

$$v_{k+1} = \overline{v}_k - \frac{I_y[I_x\overline{u}_k + I_y\overline{v}_k + I_t]}{\alpha^2 + I_x^2 + I_y^2} \tag{2.5}$$

where $\overline{u}$ and $\overline{v}$ are the average velocity vectors computed from the neighbourhood pixels of the currently pixel. For the first iteration u_0 and v_0 are set to 0 for every pixel, and the parameter α must be tuned to fit the system needs.

2.3.2 Lucas and Kanade

Lucas-Kanade proposed in [13] a gradient, local method opposing to the Horn-Schunck which is a global method. This characterization comes from the fact that it uses images derivatives and because it assumes that the flow is constant in the same local neighbourhood [19]. Thus, it is possible to estimate the flow for a pixel focusing only on the neighbourhood and ignoring the rest. This is supported by the assumption that close pixels are likely to correspond to the same object, therefore have similar velocity vectors.

Using the description in [18], and abstracting from the mathematical deduction and using the same formulation as in 2.3.1, this method estimates the flow by minimizing in a small spatial neighbourhood Ω,

$$min \sum W^2(p)[\nabla I(p,t) \times s(p,t) + I_t(p,t]^2 \tag{2.6}$$

where $W(p)$ denotes the window centred in the pixel p, which is responsible for giving more influence to the pixels closer to p. Solving equation 2.6 becomes a problem of minimization that can be solved by preforming a least squares fit, and it is given by:

$$\begin{aligned} A^T W^2 A s &= A^T W^2 b \\ s &= [A^T W^2 A]^{-1} A^T W^2 b \end{aligned} \tag{2.7}$$

where, for each pixel p_i inside the window Ω, with n pixels, at a single time t,

$$\begin{aligned} A &= [\nabla I(p_1), ..., \nabla I(p_n)]^T \\ W &= diag[w(p_1), ..., w(p_n)] \\ b &= -[I_t(p_1), ..., I_t(p_n)]^T \\ \nabla I(p_i) &= I_x(p_i), I_y(p_i) \end{aligned} \tag{2.8}$$

From equation 2.7 we obtain:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \sum_{i\in\Omega} w^2(p_i) I_x^2(p_i) & \sum_{i\in\Omega} w^2(p_i) I_x(p_i) I_y(p_i) \\ \sum_{i\in\Omega} w^2(p_i) I_x(p_i) I_y(p_i) & \sum_{i\in\Omega} w^2(p_i) I_y^2(p_i) \end{bmatrix}^{-1} \begin{bmatrix} -\sum_{i\in\Omega} w^2(p_i) I_x(p_i) I_t(p_i) \\ -\sum_{i\in\Omega} w^2(p_i) I_y(p_i) I_t(p_i) \end{bmatrix} \tag{2.9}$$

The next important step is choosing the right parameters to be used in the implementation, such as the pre-filter, derivative kernels, and the neighbourhood area. These parameters should be chosen in function of the necessary accuracy and computing power available.

2.3.3 HybridTree

The two previous methods have different characteristics, which makes each one more suitable for different images regions. Thus in [16], it is presented an HybridTree method. The optical flow computation is divided into two consecutive operations: expectation and sensing.

In the expectation phase, it is used descriptive properties of the image, such as texture gradients, dominant colour, and spatial information, to divide the image into different regions. After the image is divided into regions, it is possible to choose the most suitable method, Horn-Schunck or Lucas-Kanade, for optical flow evaluation. This dividing of the image enhances performance since avoids computation intensive tasks in uniform regions.

The decomposition of the image in the expectation phase is followed by the sensing operation, where the optical flow estimation is done individually for the many different pieces of the image. The flow estimation is done through a hierarchical combination of Horn-Schunck and Lucas-Kanade, in function of what was decided in the expectation phase.

2.4 Hardware implementation

Approaches to deal with high computation demand have been made, such as the HybidTree method explained in 2.3.3. This is a less computationally expensive method that is more suitable for systems without specialized computer devices, and which keeps at the same time good accuracy. Another option is to use dedicated hardware to implement optical flow algorithms.

Dedicated hardware can include general purpose processors (CPU), graphics processors (GPU), application specific integrated circuits (ASIC), and FPGAs. Though, the constraints related to AUV and embedded systems discard the first two hypotheses as CPU and GPU have, in general, high power consumption and need a large space. The ASIC and FPGA allow parallel implementation of many algorithms and have low power consumption. Yet, ASICs usage is only justified for large volume production because of its long and costly design, testing and production process. Besides, it is impossible to introduce any changes or improvements to the algorithm once the device is produced. The ideal solution for this problem is using FPGAs. Modern FPGA devices have similar capabilities to ASIC and are well suited for prototyping and small production. Furthermore, the main advantage is the possibility to make changes and improvements to the algorithm without changing the costly hardware components. This is the reason why FPGA devices have been used to implement optical flow algorithms such as in [20, 21, 22, 23]. Due to its parallel characteristics and the possibility of implementing a pipelined version of the algorithms, FPGAs present good results for real-time computation.

Nonetheless, due to its complexity, not all algorithms are feasible to implement in FPGAs. In order to solve this problem, most algorithms implemented in FPGA are gradient based algorithms, such as the Lucas-Kanade, which was implemented in [24, 25, 19, 26] and Horn-Shcunck in [27, 21, 20, 28].

In [24], Diaz et al., implemented an improved version of Lucas-Kanade description presented in [18], in an FPGA with a pipeline architecture of 70 stages. This implementation is able to achieve 170 frames per second with a resolution of 800 x 600 pixels with a Virtex II FPGA. In this work, the implemented systems take up 1731 equivalent gates, 10433 Flip-Flops, 760Kbits of memory and has the maximum clock frequency of 82MHz. However, it is possible to change these values by changing the image resolution or the data format and precision.

Later, in 2010, it was presented in [25] an efficient VLSI architecture for accurate computation of optical flow using Lucas-Kanade algorithm and using only fixed-point arithmetic. The architecture was simulated in a Virtex-II Pro FPGA, taking up 11086 LUTs, 23 Multipliers and uses 55 MHz clock frequency at a resolution of 640 x 480 pixels at 125 FPS.

In order to reduce the frame memory access, in [26] it was implemented a hardware architecture of Lucas-Kanade algorithm that instead of storing the original images, it stores the image after the Gaussian filtering operation and downsampling. The results of the work show that the proposed work reduces the memory access by 61%-75% relatively to [24] and [25]. This work proves that real-time processing of full HD video may be possible depending on the type of the external memory and the other parts of the system.

The work in [19] presents a parallel architecture for hierarchical estimation of optical flow, using the Lucas-Kanade algorithm with a multi-scale approach and a coarse-to-fine architecture to enable computation for large displacements. The algorithm was implemented using a Xilinx Virtex4 XC4vfx100 FPGA. With this implementation it was possible, in a mono-scale approach, to compute the optical flow at 270 fps with an image resolution of 640 x 480 pixels. In the case of multi-scale, the frame rate reaches 32 FPS with the same image resolution. The hardware resources utilization for this approach, using low resources and fixed-point arithmetic, was 4036 LUTs, 6622 Flip-Flops, 4224 Slices, and achieved a clock frequency of 83 MHz.

Kunz et al. presented in [21] an FPGA-optimized architecture for optical flow estimation based on the algorithm Horn-Schunck . The architecture presented enables the computation of optical flow for each pixel of a frame with 640 x 512 pixels at a frame rate of 30 FPS with 30 iteration of the computation, and it was able to go up to 4k resolution at a frame rate of 30 fps in its full-pipelined form with 20 iterations of the algorithm. This work also evaluates the tradeoff between throughput and hardware costs. By not saving hardware, it is possible to go up to 4k resolution and saving hardware 640 x 512 pixels. The calculation is done in fixed-point arithmetic. The board used was a Stratix IV and, for the high definition and 30 iterations, were used: 16997 LUTs, 66220 dedicated memory registers, 5.3 Mbit of BRAM, 476 DSP of 18 bit and works at a clock frequency of 294.9 MHz.

Another work using Horn-Schunck algorithm is [20]. The algorithm is implemented also in a pipelined manner being able to achieve a throughput of 175 Mpixels/s, which enables processing of Full HD video at 60 fps. The authors evaluated the impact of the numerical representation and the impact of the number of iteration on the optical flow computation. It was observed that using floating-point both angular and endpoint error decreased with the number of iterations. On the other hand, for fixed-point representation it was observed the effect of truncation if not enough bits are allocated for the fractional part, the error will increase with the number of iterations as it accumulates over time. Although, if saving resources is necessary, fixed-point representation should be used with 10 bits in the fractional part.

2.5 Velocity estimation from optical flow

Optical flow, as it was previously defined, it is the apparent motion of objects between frames. The problem is that the camera is capturing 3D motion and representing its projection in an 2D motion. This means that when the optical flow is estimated, it is obtained a velocity vector in pixels per second. Hence, the 2D motion that the camera captures must be quantified in a 3D translation and rotational movement. In [29] is shown how this conversion can be performed, briefly described in the next paragraphs.

The motion of the camera, assuming a static scene, has two components: the translational (t_x, t_y, t_z) and the rotational component (w_x, w_y, w_z). The pixel velocity is given by (u, v), f is the

focal distance of the camera, (x,y) is the position of the pixel, and Z is the height of the camera.

$$\begin{bmatrix} u \\ v \end{bmatrix} = \frac{1}{Z}\begin{bmatrix} -f & 0 & x \\ 0 & -f & y \end{bmatrix}\begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} + \begin{bmatrix} \frac{xy}{f} & -f-\frac{x^2}{f} & y \\ f+\frac{y^2}{f} & -\frac{xy}{f} & -x \end{bmatrix}\begin{bmatrix} w_x \\ w_y \\ w_z \end{bmatrix} \tag{2.10}$$

It is possible to estimate the translational velocity of the camera by doing the average of all optical flow vectors. The average operation will cancel out any rotational movement of the camera, and from that, it is possible to use that vector to compute its velocity [30] given by:

$$\begin{aligned} V_x &= Z\frac{\overline{u}}{f} \\ V_y &= Z\frac{\overline{v}}{f} \end{aligned} \tag{2.11}$$

Chapter 3

Motion field estimation

In this chapter is presented and detailed three algorithms for image motion estimation. The first one is the Lucas and Kanade (section 3.1). Then, the iterative and pyramidal version of the Lucas and Kanade (section 3.2). The last one presented is the block matching method (section 3.3). In each section the different stages of algorithms are presented and explained, the choices of the parameters are also explained. The results of each algorithm are presented in each section. It is also explained the aperture problem (section 3.4)

3.1 Lucas and Kanade

To estimate the optical flow using the Lucas and Kanade method, described in the subsection 2.3.2, we must solve the system of equation in 2.9. Through the analysis of the equation system, it is possible to deduct the flow chart presented in the figure 3.1.

The first stage after the frames $t-1$ and t are in memory is the spatial Gaussian smoothing stage (subsection 3.1.1), where a Gaussian filter is applied to the frames. With the smoothed frame t it is computed the image gradient (vertical and horizontal derivative) (subsection 3.1.2). The frame $t-1$ is used with the frame t to compute the temporal derivative which will be further smoothed with another stage of the Gaussian smoothing to remove any noise left (subsection 3.1.3).

After the different derivatives are computed, the next stage (subsection 3.1.4) is creating the matrices $[A^T W^2 A]$ and $[A^T W^2 b]$ from the equation 2.9. After having the matrices created using the integration window Ω, the system of equations can be solved in the stage "Matrix solver". At this point, the system can be solved with different approaches that will be presented in the section 3.1.5. As this is a dense optical flow method, these last two stages have to be repeated for every pixel in the image.

3.1.1 Spatial Gaussian smoothing (stage G1)

The first stage, after the image acquisition, is to perform a Gaussian blur in the captured frame which helps to improve the Lucas-Kanade method accuracy. The Gaussian blur operation consists

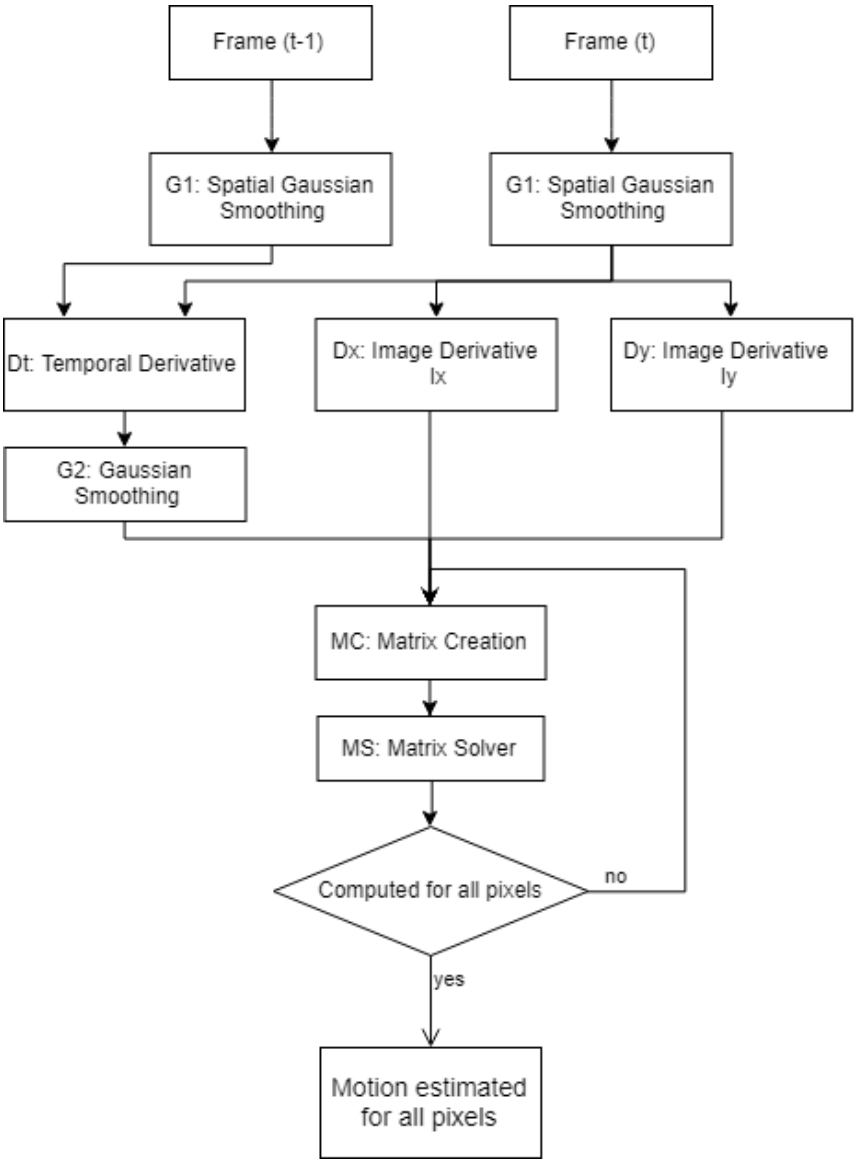


Figure 3.1: Flow chart of the process to compute the optical flow using the Lucas and Kanade.

in the blurring of the image by a Gaussian function (equation 3.1) which will smooth out the image, reducing noise and very sharp edges.

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{3.1}$$

The Gaussian distribution is shown in figure 3.2.

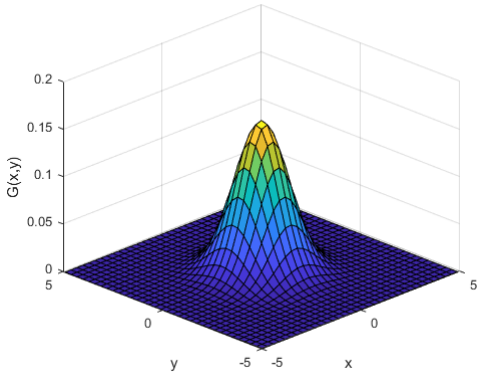


Figure 3.2: Gaussian distribution with a standard deviation (σ) of 1 and mean $(0,0)$.

Since the image is stored as a collection of discrete pixels it is necessary to do a discrete approximation of the Gaussian function. This can be achieved by sampling the Gaussian function at discrete points, as the midpoint of the pixel. Although, when doing the conversion from continuous to discrete, the sum of the values will be different than 1 which will result in the darkening or brightening of the image. This can be solved by normalizing the values, dividing all terms in the kernel by the sum of all values. Another problem is that for small kernels, sampling at the pixel midpoint leads to large errors. It is possible to reduce this error by integrating the Gaussian function over the pixel area.

However, in order to simplify and facilitate hardware implementation, by elimination of the expensive division operation, it was chosen to use a binomial distribution [31] with the probability of 0.5 instead of the Gaussian function 3.1 as it was done in [26]. The different sizes of the simplified Gaussian kernels are shown in table 3.1.

Table 3.1: From left to right: 3x3, 5x5, 7x7 simplified Gaussian kernels

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} /16 \quad \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix} /256 \quad \begin{bmatrix} 1 & 6 & 15 & 20 & 15 & 6 & 1 \\ 6 & 36 & 90 & 120 & 90 & 36 & 6 \\ 15 & 90 & 225 & 300 & 225 & 90 & 15 \\ 20 & 120 & 300 & 400 & 300 & 120 & 20 \\ 15 & 90 & 225 & 300 & 225 & 90 & 15 \\ 6 & 36 & 90 & 120 & 90 & 36 & 6 \\ 1 & 6 & 15 & 20 & 15 & 6 & 1 \end{bmatrix} /4096$$

The standard deviation (σ) can be calculated for each kernel by $\sigma = \sqrt{np(1-p)}$, being p the probability and n the kernel size minus one. This yields 0.707, 1 and 1.225 respectively as standard deviations of the kernels in the table 3.1. In the figure 3.3 is possible to observe the effects of the kernels convolution with the image.

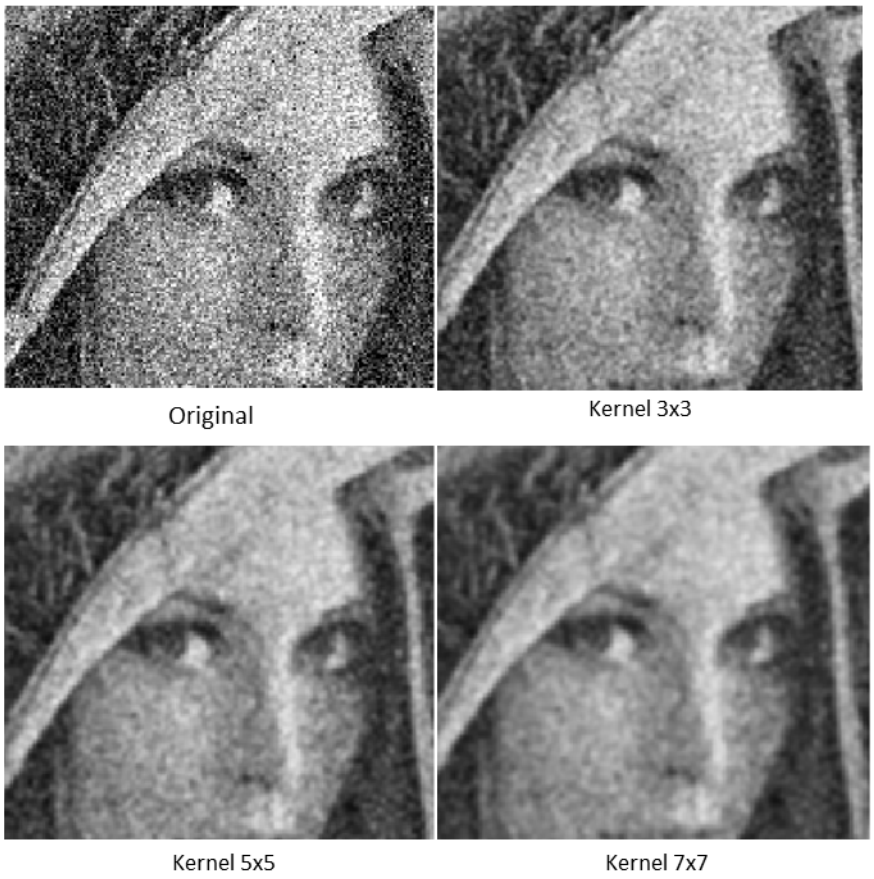


Figure 3.3: The original image has Gaussian noise added. In the other images is possible to observe that this noise is more attenuated as the filter size increases however, some of the fine details of the image are lost.

3.1.2 Spatial derivatives (stages Dx and Dy)

The gradient of an image is the direction of change in the intensity of the image. As the image is defined in a 2D plane, the gradient of an image has two components: one in the x direction, the horizontal derivative, and one in the y direction, the vertical derivative. At each point the gradient vector points in the direction of the largest possible intensity increase, and its magnitude corresponds to the rate of change in that direction.

The image derivatives work as a high pass filter, removing the mean brightness component of the image and highlighting the changes in brightness. Thus, the pixels where the derivative of the image is large will be edge pixels, this is, pixels that correspond to edges of textures or objects.

To calculate the image derivatives, in this work was decided to use the Simoncelli's kernels. In [32] it is demonstrated that this kernel has the best results. To compute the I_x it was used the kernel $[-1\ 8\ 0\ -8\ 1]/12$, and to compute I_y its transpose. The result of the image 2D convolution with the kernel can be observed in the figure 3.4

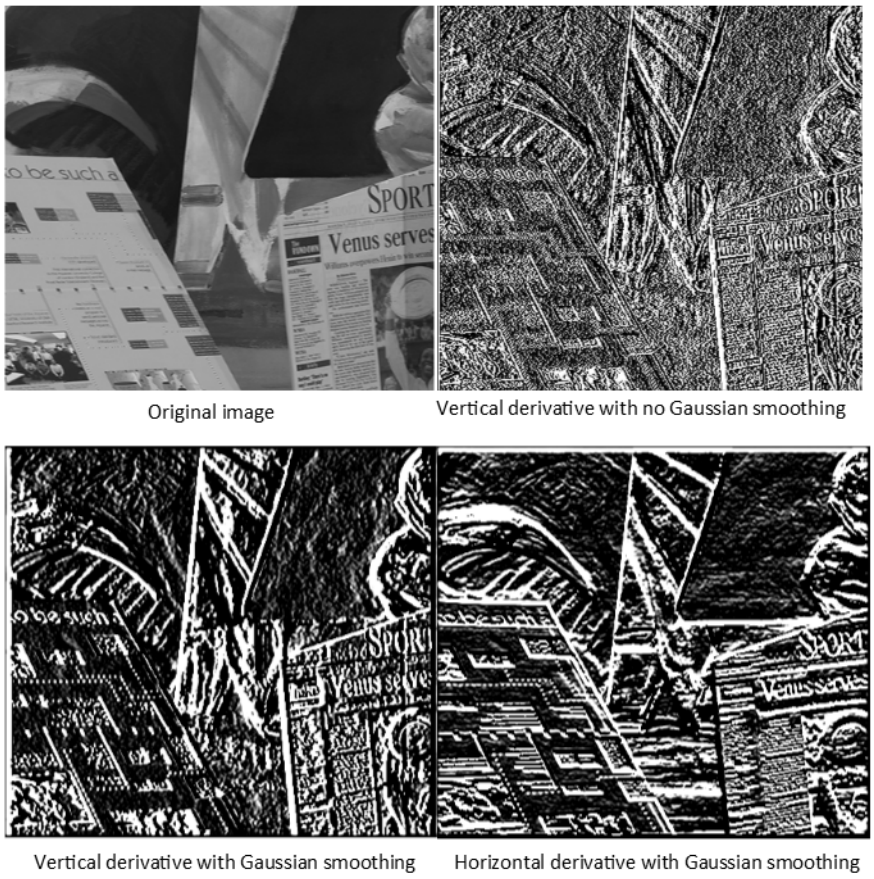


Figure 3.4: Results of the gradient computation. In the top right is the x derivative computed without previous Gaussian smoothing and in the bottom are de x and y derivatives computed with previews Gaussian smoothing.

In the figure 3.4 we can observe how important is the Gaussian smoothing stage. In the top right image we observe high amounts of noise but when the gradient is computed with an image previously smoothed, in this case with the 5x5 kernel from the table 3.1, the noise is drastically reduced which will further improve in the optical flow computation.

3.1.3 Temporal Derivative (stages Dt and G2)

The temporal derivative consists in calculating the difference between two consecutive frames, i.e. it is subtracted from the frame at the instant t the frame at the instance $t-1$. From this calculation we obtain the regions of the image that moved between the two frames, as can be seen in figure 3.5.

By observing of the figure 3.5, it is possible to conclude that the two phases of Gaussian smoothing, one before computing the derivatives and another smoothing stage after computing the Dt, yields the better results for the temporal derivative and with significantly less noise. The Gaussian kernel used was the 5x5 kernel from the table 3.1. The 5-by-5 Gaussian kernel was chosen as it has a good balance between computation requirements and results.

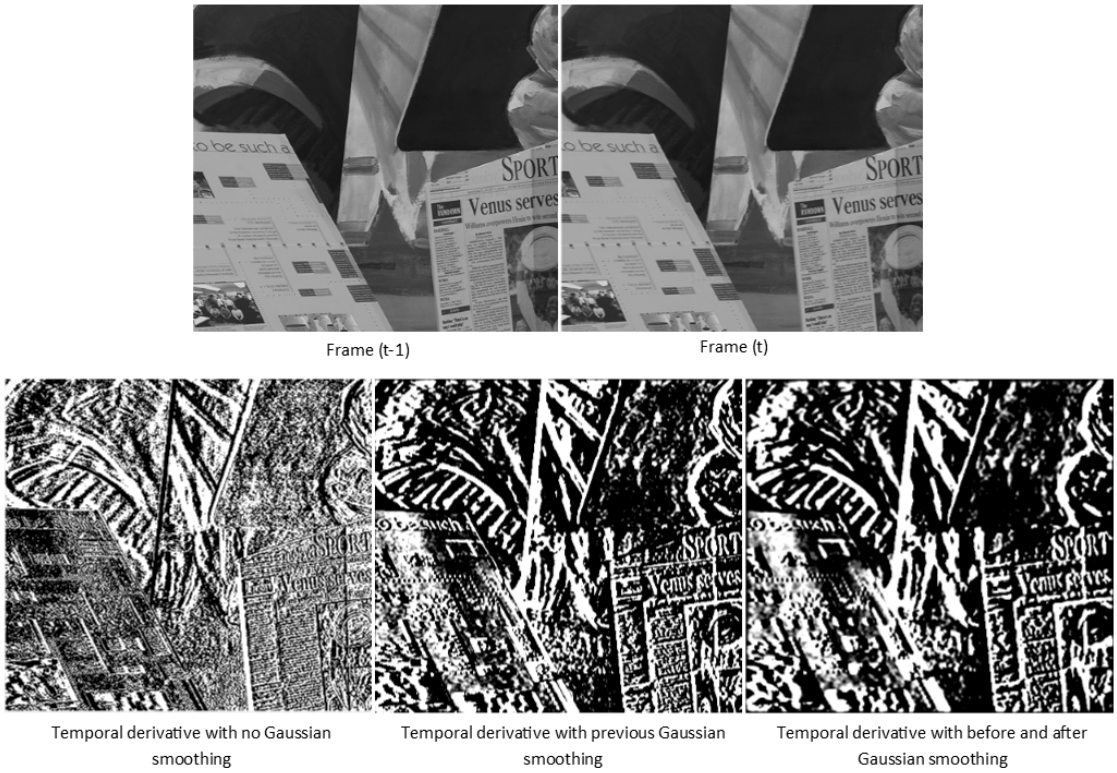


Figure 3.5: Result of the temporal derivative of the image. The bottom left image is the derivative calculated without any steps of the Gaussian smoothing, in the bottom centre is the derivative computed with the previous images smoothed, and in the bottom right is the derivative computed with the two phases of Gaussian smoothing.

3.1.4 Matrix Creation (stage MC)

For every pixel, it is necessary to solve the system of equations from 2.9. As it was explained previously, the Lucas and Kanade method uses the concept that pixels in the same neighbourhood are likely to correspond to the same object, therefore moving with the same velocity. Thus, in practical terms, looking just at the pixels in a neighbourhood of size w it is possible to compute the optical flow for each pixel. It also makes sense to not give the same importance to all neighbourhood, as the pixels that are further away are less likely to move with the same velocity as pixels that are closer to the central pixel, for which the velocity is being computed. Thus, it is used a weighting matrix W with the coefficients of the Gaussian kernel 5x5 of the table 3.1.

Choosing the right size for the neighbourhood is important. On the one hand a larger window will be able to handle large motions; however, details will be lost, as the assumption that the pixels inside the same window belong to the same object is violated much more frequently, as pixels can have different velocities. Moreover, computing time will escalate very rapidly. On the other hand, smaller windows will yield results with more details and lower computing time, but will be unable to handle large motions, which will produce erroneous results.

Solving the optical flow using Lucas and Kanade (2.9) is solving a system of the type $V = G^{-1}B$, where $[A^T W^2 A]$ is G and $[A^T W^2 b]$ is B

The process for creating the matrices G and B is described as follows:

1. Set the size of the $halfWindow$ to the floor of $windowSize/2$, $windowSize$ is always even.
2. Assuming the position at the pixel (i, j), with $(i >= halfWindow)$ and $(i <= ImageVerticallSize - halfWindow)$ and $(j >= halfWindow)$ and $(j <= ImageHorizontalSize - halfWindow)$.
3. Get the neighbourhood for the pixel (i, j) from I_x, I_y and I_t.
   $lr = i - halfWindow$
   $hr = i + halfWindow$
   $lc = j - halfWindow$
   $hc = j + halfWindow$
   $tempI_x = I_x(lr : hr, lc : hc)$
   $tempI_y = I_y(lr : hr, lc : hc)$
   $tempI_t = I_t(lr : hr, lc : hc)$
4. **for** every pixel (m, n) in the window.
   $G(1,1) = G(1,1) + tempI_x(m,n) * tempI_x(m,n) * W(m,n);$
   $G(1,2) = G(1,2) + tempI_x(m,n) * tempI_y(m,n) * W(m,n);$
   $G(2,1) = G(1,2);$
   $G(2,2) = G(2,2) + tempI_y(m,n) * tempI_y(m,n) * W(m,n);$

   $B(1,1) = B(1,1) - tempI_x(m,n) * tempI_t(m,n) * W(m,n);$
   $B(2,1) = B(2,1) - tempI_y(m,n) * tempI_t(m,n) * W(m,n);$

Note that the $G(2,1)$ value is equal to $G(1,2)$.
The matrices G and B are then passed to the next step.

3.1.5 Matrix Solver (stage MS)

To solve the 2-by-2 linear least square estimate ($V = G^{-1}B$), from the equation 2.7, it was chosen to follow the work done by Barron et al. in [18], as this work presented good results for this approach. First are computed the eigenvalues of the matrix $G = \begin{bmatrix} a & b \\ b & c \end{bmatrix}$:

$$\lambda_1 = \frac{a+c}{2} \pm \frac{\sqrt{4b^2 + (a-c)^2}}{2}; i = 1,2 \tag{3.2}$$

The eigenvalues are then compared to the a threshold τ, that corresponds to a threshold for noise reduction, and the velocity is estimated using one of the following methods:

1. If $\lambda_1 \geq \tau$ and $\lambda_2 \geq \tau$
   G matrix is nonsingular and the system of equations is solved using Cramer's rule, computing the inverse matrix of G.

$$\frac{1}{det(G)}\begin{bmatrix} c & -b \\ -b & a \end{bmatrix}\begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} u \\ v \end{bmatrix} \tag{3.3}$$
$$det(G) = ac - bb$$

2. If $\lambda_1 \geq \tau$ and $\lambda_2 < \tau$
   Matrix G is singular and non-invertible, the gradient is normalized to calculate (u,v). In [29], the authors calculated the normal velocity component to the gradient through the equation:

$$\begin{bmatrix} u \\ v \end{bmatrix} = -\frac{I_t}{||\nabla I||^2}\nabla I \tag{3.4}$$
$$\nabla I = \begin{bmatrix} I_x \\ I_y \end{bmatrix}$$

3. If $\lambda_1 < \tau$ and $\lambda_2 < \tau$
   It is impossible to solve the system thus (u,v) is $(0,0)$

As the objective is to simplify the optical flow computation, it becomes apparent that solving equation 3.2 is very computing intensive. The author of [29] suggested that instead of using the eigenvalues of the matrix G, it should be used the determinant of the matrix and the norm of the gradient to decide which of the above methods to apply. If the determinant is above 1, it is used the case 1, else if the square of the norm is above 1, it is used the case 2, else both are below 1, it is used the case 3.

In the figure 3.6 is presented the results of the Lucas and Kanade computation following the scheme presented here.

The values for τ are the ones used by Barron et. al. in [18], and when using the 5x5 neighbourhood the W matrix has the coefficients of the 5x5 Gaussian kernel from table 3.1. Alternatively, when using the 11x11 neighbourhood it was used a kernel with all coefficients equal to one. In the two top right images in figure 3.6, it is observable that using the 5x5 weights matrix the optical flow is more localized in the object edges, in the scene, and does not represent the overall movement as good as using the 11x11 weights matrix (the bottom three images).

A comparison was made between the use of different values for τ being that the most notorious difference is between the two top right images. The optical flow computed with τ equal to 1.0 has more noise, as expected, and it is computed in points where the confidence is low. On the other hand, the optical flow computed with τ equal to 5.0 has less noise and it is more accurate. In the bottom images, there is no observable difference between the different values used for τ. Furthermore, when an 11x11 neighbourhood is used, there is no observable difference among the

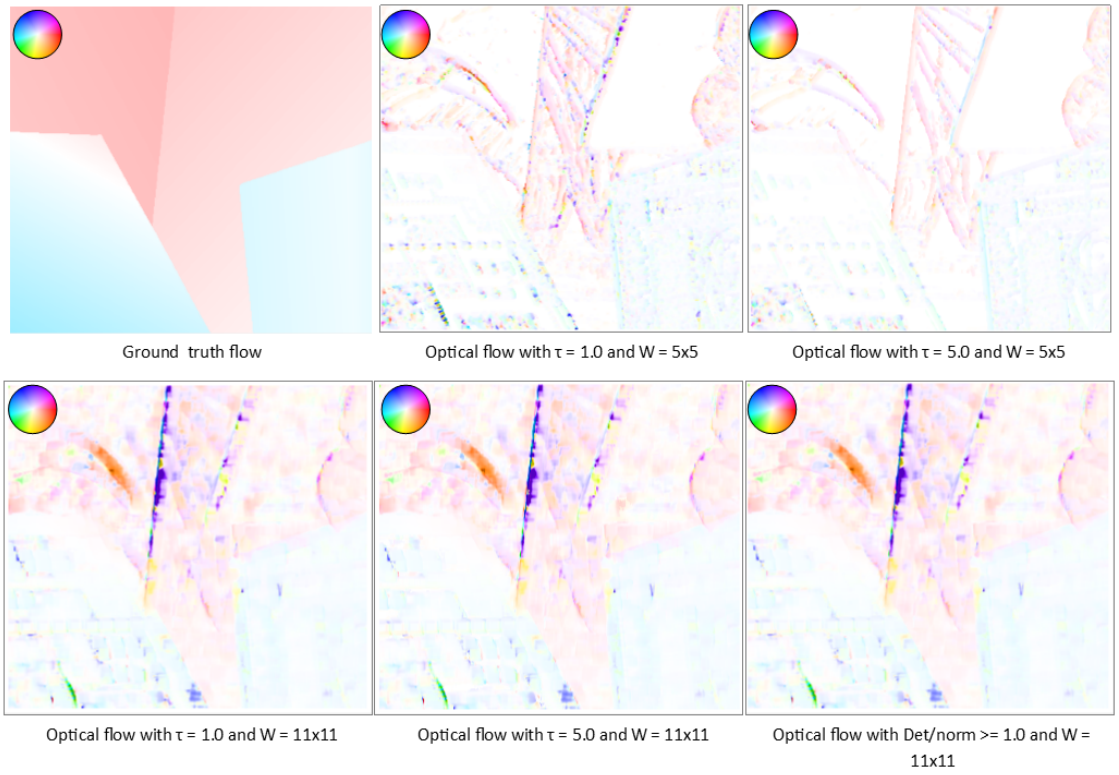


Figure 3.6: Results of optical flow computation with different values for τ and different sizes of the neighbourhood. The top left image is the ground truth flow for the venus sequence from figure 3.5.

usage of the eigenvalues or the usage of the determinant and norm as deciding values to where to compute the optical flow.

3.1.6 Results

When considering small movements, the algorithm presented is able to compute correctly the optical flow; However, when the movement is large it starts to fail. Looking at the figure 3.7, when the optical flow is computed using a neighbourhood of size 11x11, the algorithm was able to compute the velocity of the two slow-moving cars (light blue) correctly and with low noise. Nevertheless, it was not able to do the same with the fast moving cars, calculating the optical flow with large amounts of noise. Even though, it is possible to see that in some regions the velocity is right (red), this is not a reliable measurement. One possible way to overcome this problem is by enlarging the neighbourhood size, as shown in the figure 3.7 in which the window size goes up to 41x41. Using this window size, the noise in the fast moving cars is almost gone and the optical flow field from the slow-moving cars is more homogeneous. However, even though the direction of the fast moving cars is correctly computed, the magnitude of the velocity is wrong. The cars are moving at a higher velocity than the one computed, as result the colour should not be a light red but a darker red. Not just computing the direction, in high displacements, is important but the

magnitude is also of utmost importance for the final application, as the perceived motion of the AUV can be very high between consecutive frames.

Not being able to compute the velocity properly for large displacements makes this version of the algorithm, as is, unsuitable for our goal. A version of the Lucas and Kanade algorithm that is able to compute the optical flow field correctly for large displacements is the pyramidal and iterative version described in [33], or a block matching algorithm. In the following sections, these implementations will be discussed.

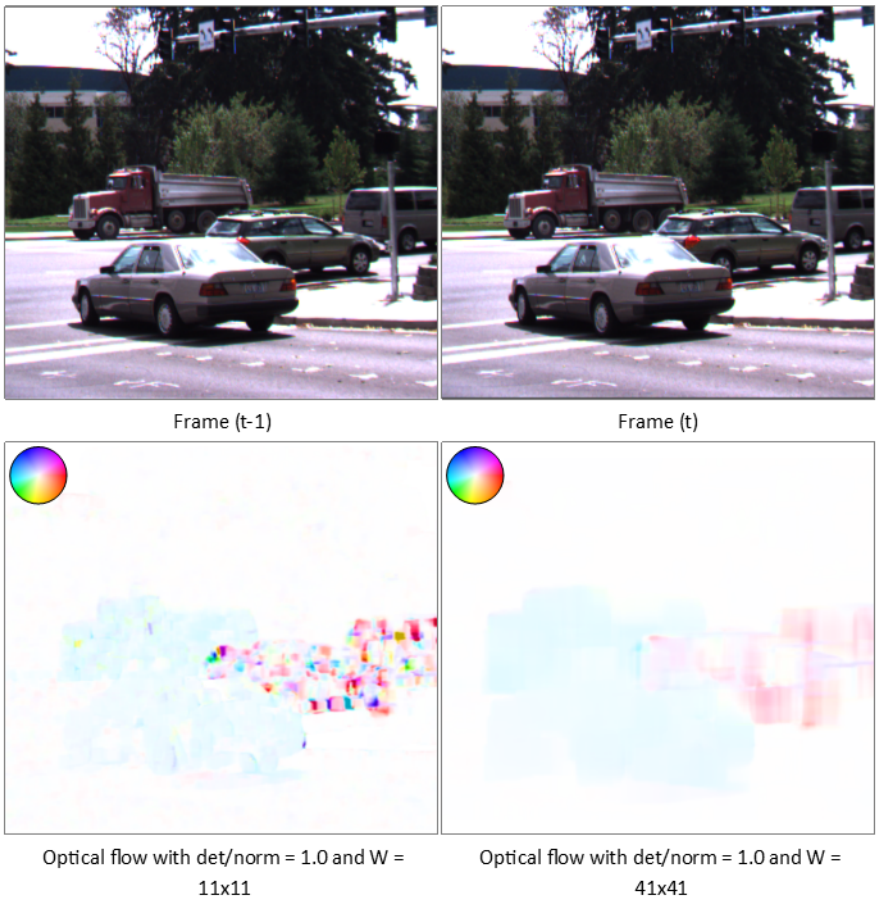


Figure 3.7: Demonstration of the maximum velocity of the optical flow. The dump truck and the car in the foreground are moving slowly and the other two cars are moving at a higher speed.

3.2 Iterative and pyramidal Lucas and Kanade

One simple way to make the apparent motion of an object smaller is by reducing the frame size. For example, a 10 pixel motion in a frame of 640x480 becomes a 5 pixel motion is a frame of 320x240. The pyramidal Lucas and Kanade consists in making the apparent motion smaller by creating an image pyramid, in which the first level of the pyramid is the original image and the other levels are subsampled images, each one with half the size of the previous level. Computing

the optical flow consists in, starting at the last level, compute the optical flow as described in the previous section. Then, when the value of the optical flow is computed it is passed to the next level, where the image is warped and the optical flow is computed again. This procedure is repeated until the first level of the pyramid. In each level of the pyramid the optical flow is computed iteratively, this means that the optical flow is computed then, the image is warped based on the value calculated for the optical flow, and the optical flow is computed again, increasing the accuracy at each iteration, this can be better understood in the algorithm description in 1.

Based on the pyramidal implementation of the affine Lucas and Kanade in [33] for features tracking, in this section, it is presented a modified version for a dense optical flow that is possible to implement in hardware with the accuracy requirements for our final application.

input : Frame at time $t-1$, I, and frame at time t,J.
output: Optical flow field (u,v)

```
1  Build pyramidal representation of frame J and I. Frames I(L) and J(L) for each level L of
     the pyramid;
2  Initialization of the pyramidal guess, (u,v) field equals to 0;
3  for L = nPyramidLevels to 1 do
4      Compute the I_x and I_y derivative of the image I(L);
5      Resize (u,v) field to the current pyramid level and multiply by 2;
6      foreach Pixel in the image J(L), (px,py) do
7          Get window of J(L), centred in the current pixel;
8          for k = 1 to nIterations do
9              Warp frame:
10                 Get offsetted window of the derivatives I_x(k) and I_y(k);
11                 Get offsetted window of the frame I(L,k);
12             Compute the I_t(k) derivative;
13             Construction of the G and B matrix;
14             Solve the least square system;
15             Update guess for the next iteration of the algorithm;
16         end
17     end
18 end
```

Algorithm 1: Algorithm description of the iterative pyramidal approach for Lucas and Kanade

The algorithm can be divided into different stages.

- **Pyramid construction and initialization of the algorithm** - lines 1 and 2 - Description of the pyramid construction and the initialization of the variables of the algorithm. Will be discussed in the subsection 3.2.1.

- **Spatial derivatives and optical flow field resize** - lines 4 and 5 - Details of the computing the spatial derivatives and resizing for the next level of the optical flow field. Subsection

3.2.2.

- **Image warp** - lines 9, 10 and 11 - Warping of the image and its derivatives using the current optical flow field. Subsection 3.2.3.
- **Temporal derivative** - line 12- Computation of the temporal derivative.
- **Optical flow computation** - lines 13, 14 and 15 - optical flow is computed by the creating the G and B matrix and solving the system as described previously in 3.1.5. Subsection 3.2.4.

3.2.1 Pyramid construction

The images are scaled down by constructing a Gaussian pyramid. The construction of this kind of pyramid consists in the smoothing of the image with an appropriate Gaussian kernel, in this case, a 5 by 5 Gaussian kernel is used because a 3 by 3 would be too small, and not include sufficient information, and a 7 by 7 too large without visible gains. Then, the smoothed image subsampled by a factor of 2 in the x and y direction. The resultant image is subjected to the same process and this is repeated, as many times as we wish. Each cycle results in a smaller image with increased smoothing and decreased sampling density, that is, less image resolution. If these images were represented graphically, they will look like a pyramid.

It is mandatory to perform the Gaussian smoothing due to aliasing, which appearers when the sampling frequency is insufficient to capture the changes in the signal. In Gaussian pyramids, this will always happen as subsampling is done at half the pixels at each level. As smoothing the image will reduce the maximum frequency of the image, removing the fast changes that the subsampling would miss, this operation is important to be performed before downsampling. In the figure 3.8 is the representation if the aliasing effect on the images, the right image has lost its smoothed transitions due to aliasing however, the left image, although it has lost some details it has preserved its smoothed transitions.

3.2.2 Image derivatives and optical flow resize

The process to compute the image gradient is the same as described in the subsection 3.1.2. In this case, for each level of the pyramid, the gradient of the image $I(L)$ is computed. In the last level and in order to reduce the noise of the derivatives, the image is subjected to a Gaussian smoothing before computing the derivative.

As we go through the levels of the pyramid, starting in the first level, the (u,v) field has to be resized to the size of the current level of the pyramid. The chosen resizing process corresponds to the nearest neighbour method which copies the value (u,v) of the nearest pixel to the current pixel. This method is accurate enough, as we do not need high definition in the final result, and it is simpler than interpolation, for hardware implementation. Another necessary operation is to multiply the optical flow field by 2. The reason for this operation is that on two sequences, in which one is the twice the size the other, the apparent movement will also be twice the size.

Figure 3.8: Figure demonstrating the aliasing effect (right image) and in the left image is the same figure but with the smoothed operation being performed before downsampling. Both figures are 48 by 48 pixels.

3.2.3 Image warp

The warp operation consists in getting the the window, $I_x(k)$, $I_y(k)$ and $I(L,k)$ for the construction of the matrix G and B. In this operation, to the window will be added an offset of the current value of the optical flow for the pixel being computed. This operation will approximate at each iteration the frame I to the frame J.

When warping the frame we must have into account the size of the frame. When close to the border, the window can expand beyond the image boundaries when offset by the current value of the optical flow. To prevent this from happening, the optical flow computation is stopped for that pixel when it is detected that the window is out of boundaries.

As the image being warped is the I frame, and the warping approximates the frame I to the frame J, the offset is computed by subtracting the current optical flow value for the pixel, in the image J, being computed. It is important to remember that the optical flow is the displacement between the frame I and J.

3.2.4 Computing the optical flow

The temporal derivative is computed by subtracting the current window of the image $I(L,k)$ to the current window of the image $J(L)$, centred in the pixel for which the optical flow is being computed, $I_t(k) = J(L) - I(L,k)$.

The construction of the matrices G and B is then performed similarly to what is described in the section 3.1.4 point 4. A window size 11 by 11 was chosen, giving the same weight to all the pixels. Then, the optical flow is computed by the method described in 3.1.5, using the determinant of the matrix as well as the squared norm of the gradient as deciding factors.

Another aspect that needs to be addressed carefully is the computation of the optical flow when close to the boundary of the images. When computing the optical flow for these points, part of the integration window can fall outside the image. This observation becomes important

when the number of pyramid levels is large. If the integration window of size $(2w+1)$x$(2w+1)$ is enforced to be always completely inside the image, there will be a "forbidden band" of size w around the image, of the current pyramid level, where the optical flow cannot be computed. If L is the pyramid height, the effective forbidden band around the original image will be of $2^L w_x$. If we take a look at the numbers, for $w = 5$ and $L = 3$ we have a "forbidden band" of 40 pixels around the original image. This means that we are not able to track the pixels that are inside that band.

3.2.5 Results

The results obtained by this method are significantly better than the simple Lucas and Kanade. The pyramid approach has the ability to handle large motions between frames, and the iterative refinement improves the optical flow accuracy. In order to obtain good results is crucial to decide how many levels of the pyramid, and the number of iterations in the algorithm. Hereafter, it will be presented the results for different values of these parameters.

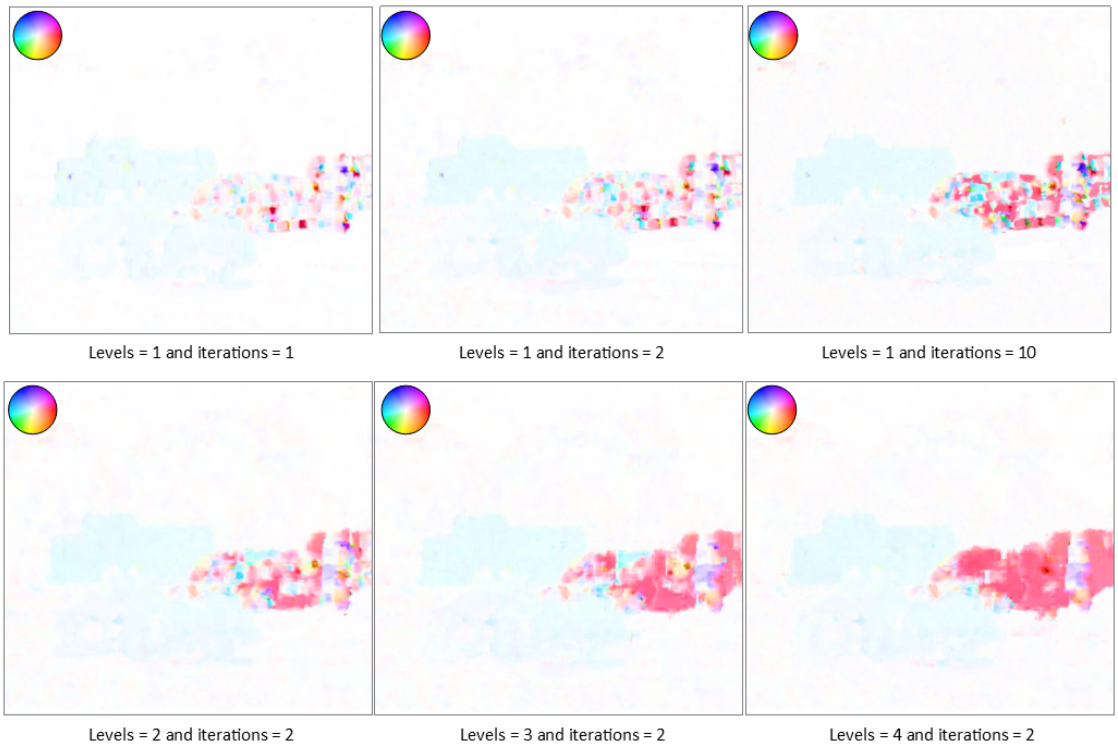


Figure 3.9: Results of the application of the method described for the sequence of the dump truck in the figure 3.7. The different results are obtained by variation of the number of levels in the pyramid and number of iterations.

Observing the sequence of images in the first row of the figure 3.9, as the number of iterations increases the optical flow values converge to the real displacement. However, this is only true for the small portions of the image where the direction of the displacement was computed correctly (there should be two cars in blue and two cars in red). In the second row of the figure, keeping

the number of iterations constant and equal to two, it was evaluated the effect of the number of levels in the pyramid. As the number of levels increase, the algorithm is able to reduce noise for the optical flow of the fast moving cars, as the problem of large motions is minimized.

3.3 Block matching

An alternative to optical flow for estimating the image movement is block matching, which consists in finding matching macro-blocks in a sequence of frames with the purpose of estimating the motion between frames. As in optical flow, the assumption behind the motion estimation using block matching is that patterns corresponding to the same object do not change between frames. The figure 3.10 represents this process, where the $M \times M$ search window is where the block $N \times N$ in the current frame will be searched.

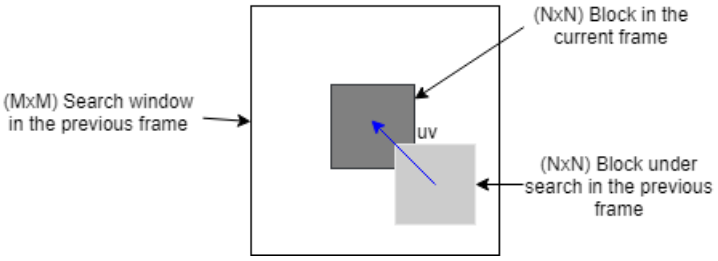


Figure 3.10: Figure of the block matching algorithm.

The block matching algorithm involves dividing the current frame into macroblocks ($N \times N$ block) and then comparing that block to all the other possibles blocks inside the window $M \times M$ of the previous frame. As a result, a vector is created representing the movement of the block from one frame to the other frame.

The size of the search window in the previous frame, M, defines the size, in pixels, of the window where the block will be searched. As the window size increases also the possibility of finding a good match increases. Nevertheless, increasing the size of the search window makes the search of the blocks computationally expensive. Moreover, the size of the block, N, has to be sufficiently large to include enough features preventing false correspondences. In this case, it was defined that the size of the block is half the size of the search window. Taking into consideration this parameter, the maximum vertical and horizontal movement that can be estimated correctly by the block matching process is $\frac{M}{4}$.

The algorithm implemented is described in the figure 3.11. The first stage is to smooth the frames with a 7x7 Gaussian Kernel from the table 3.1. This step removes noise, which will improve the block matching process. Then, the smoothed frames are divided; the frame (t-1) is divided into search windows and the frame (t) is divided into blocks to be searched in the previous frame. In the block matching stage is performed a *brute force* comparison between the block of the frame (t) and all the possible blocks of the frame (t-1). There is a block matching stage for each block of the frame. The optical flow vector is the difference in position between the block in the frame $t-1$ and the block t.

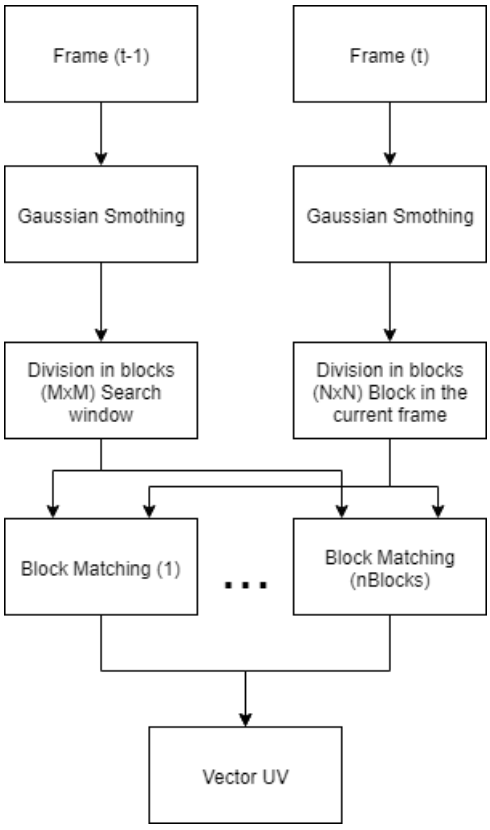


Figure 3.11: Block matching flow diagram.

The metric for matching the block in the search window with the block that is being searched is:

$$Mean\ Absolute\ Difference = \frac{1}{N^2}\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}|P_{ij} - I_{ij}| \tag{3.5}$$

Where N is the size of the block being searched, P_{ij} and I_{ij} are the pixels being compared between the block and the search window.

3.3.1 Results

Using the venus sequence from the figure 3.5 and the dump truck sequence from the figure 3.7, the algorithm described was tested and the results are presented in the figure 3.12.

Observing the results from the figure 3.12, it can be concluded that with this algorithm it is, naturally, not possible to compute the optical flow with high density. The number of divisions, and as consequence the resolution of the optical flow, are limited by the maximum velocity. In the venus sequence is shown that with an 8×8 division it is already possible to observe the blue part and the red one. Importantly, as the number of divisions increases and the size of the search window decreases the resolution increases without significant losses of quality. In the dump truck sequence, in the first image, from left to right, the slow-moving cars and the fast-moving cars

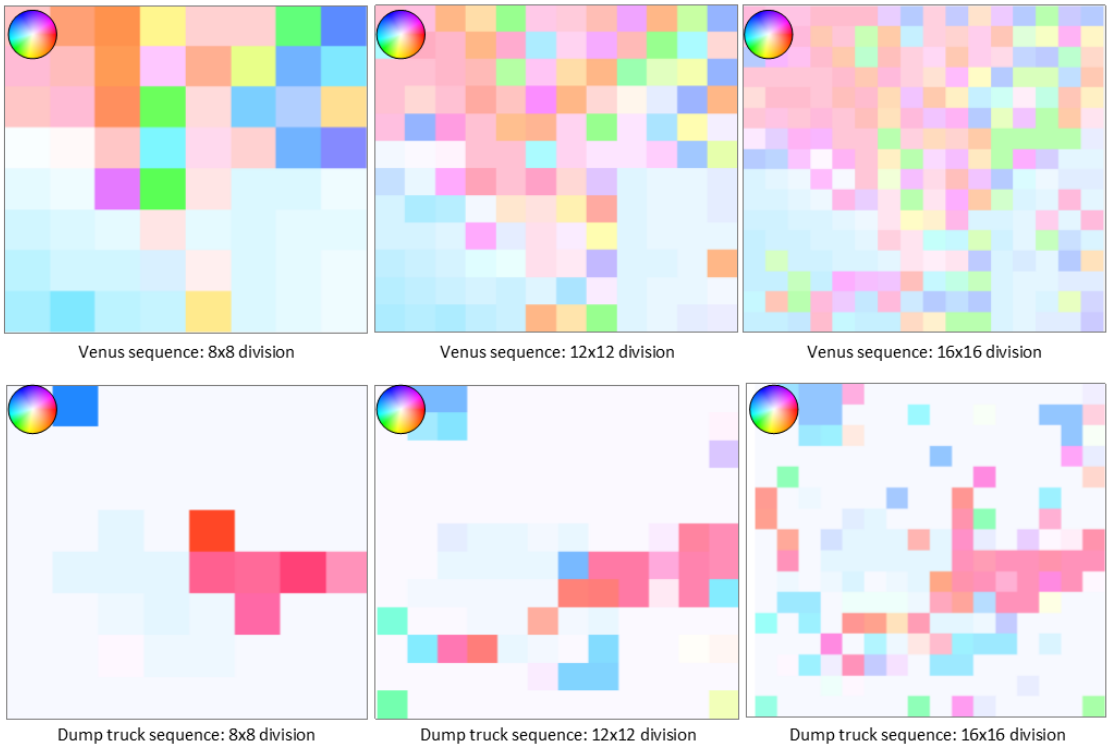


Figure 3.12: Results of the block matching method. In the first are the results for the venus sequence, in the bottom row are the results of the dump truck sequence

are identified and the magnitude of their velocity is correctly computed. However, this is a low-resolution computation of the optical flow. Moreover, as the division is increased the quality of the resultant optical flow decreases too. Although having high density in the optical flow computation is important to see the detailed movement of the objects, for our final application the simple use of an 8×8 division can be enough.

3.4 Aperture problem

The aperture problem is defined in [29] as *"the inability to measure or to fully estimate motion in regions of the image that do not exhibit distinguishable characteristics"*. This means that in the figure 3.13 the true motion direction cannot be estimated correctly based on the visual input of the aperture, which "looks" at the world through a finite opening. In the figure 3.13 below, the black bar is moving down and to the right, but the movement detected will be only down unless at least one corner of the bar becomes visible. This problem is not only inherent to the optical flow computation but it also happens with the motion perception of animal and Humans, as an optical illusion.

In the figure 3.13, is only possible to compute the motion in the gradient direction, which is the perpendicular direction to the black bar. To compute the optical flow value, normal to the gradient, it is used the equation 3.4.

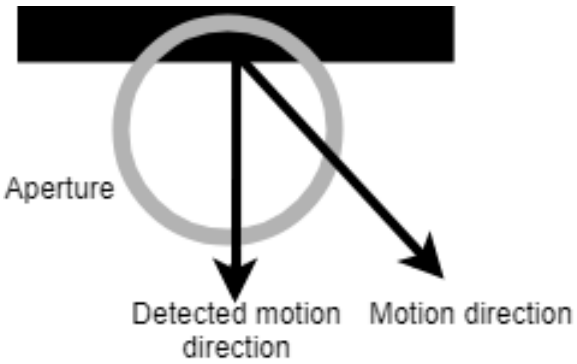


Figure 3.13: Representation of the aperture problem. The true motion direction can not be computed because the aperture problem therefore, only the normal motion to the gradient can be computed.

Chapter 4

Velocity estimation from motion field

Throughout this chapter it is analysed how to extract information about the AUV's velocity from the optical flow field. Firstly, it is presented a brief description of the problems related to capturing a 3D motion with a monocular camera (section 4.1). In addiction, it is also described the parallax effect of motion perception. Next, it is presented a description of how to estimate the translational velocity of the AUV in the x and y direction, using optical flow computing methods previously described (section 4.4). Finally, It will also be presented a methodology on how to estimate the rotational velocity of the AUV, w, in figure 4.1 (section 4.5). At last, is presented and discussed the experimental results of the tests performed (section 4.6).

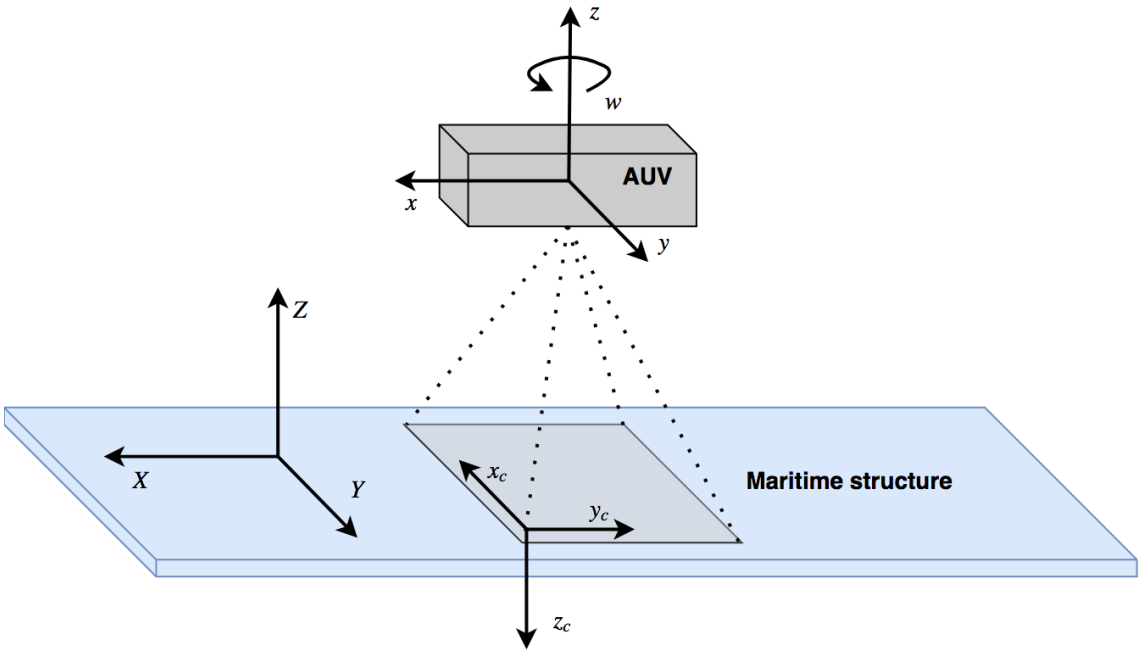


Figure 4.1: AUV reference frame and world frame.

4.1 Introduction

Estimating the AUV velocity is a challenging task because the tree-dimensional motion of an object in the world plane, considering a monocular camera, becomes a two-dimensional projected motion in the camera plane. This means that the depth of the movement is lost. Moreover, the movement of an object has two components, a translational and a rotational. The mathematical description of the optical flow field from a 3D motion is given by the equation 2.10.

The optical flow methods described before yield the displacement, in pixels, between two frames of a video. This displacement can be easily converted to velocity by dividing it by the frame period ($1/FPS$), as in the equation 4.1.

$$V \ \text{pixel/s} = \frac{(u,v)}{\frac{1}{FPS}} \tag{4.1}$$

From the equation 2.11 it is possible to translate from the pixels velocity to the velocity in the reference frame of the AUV. Also, we can observe that the (u,v) velocity is inversely proportional to the distance from the camera and this is known as the parallax effect (see section 4.1.1).

The figure 4.2 presents a flow diagram with the steps taken to retrieve the AUVs velocity from the optical flow field. The first step is dividing the optical flow field in subsections to ease the computation in the following steps and also to filter erroneous vectors, this procedure is explained in the section 4.2. The next step is a low pass filter in the time domain, which smooths the apparent movement of the AUV and eliminates high frequency noise, further details in section 4.3. After, it comes the estimation of translational velocity and in parallel the estimation of angular velocity, section 4.4 and 4.5, respectively.

4.1.1 Parallax effect

The parallax effect, as defined in [29], is the *"inability to distinguish between a near object that moves slowly from a distant object that moves quickly, and vice versa, if the the camera moves and the object remains static on the environment"*. This means that if two stationary points, at different depths, are being observed by an observer moving in the environment, the apparent motion of the to points will be different due to parallax effect. For this reason it is required, in case of using monocular systems, to resort to an depth sensor.

4.2 Dividing optical flow field and spatial averaging

This is the first step to compute the angular and translational velocity of the AUV. In this operation the flow field is divided in smaller areas and the erroneous vectors are filtered out. As the AUV camera is looking down, the velocity computed is relative to the seabed or another rigid structure that should take all, or at least most of the camera vision field, Therefore, all the values for the optical flow, for pure translational movement, should be all in the same direction. However, we have to take into account that sometimes small maritime structures, such as sea animals, will be

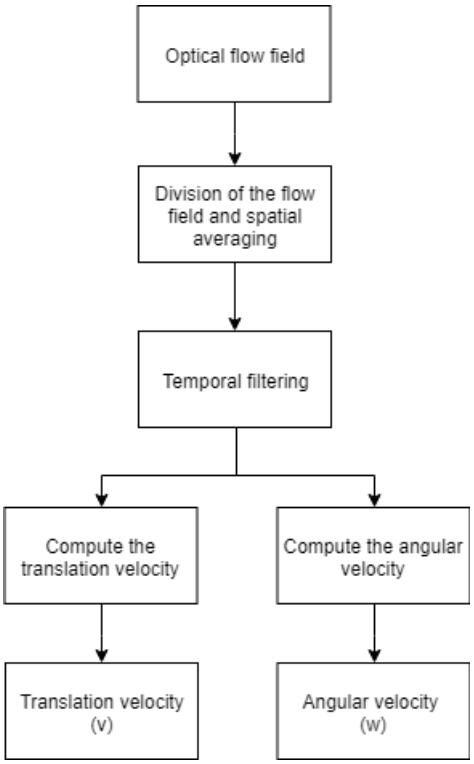


Figure 4.2: Flow diagram for angular and translation velocity estimation.

captured by the camera. As they are moving relatively to the seabed that will create regions in the optical flow field that are not aligned with most of the flow field. Furthermore, due to the parallax effect, there will be structures that are close the camera and appear to move faster and vice versa. Another problem is the rotational movements of the AUV that will create distinct optical flow fields. Moreover, the movement of the camera approximating and moving away will generate another type of optical flow fields. These different types of motion fields and camera motion are represented in the figure 4.3

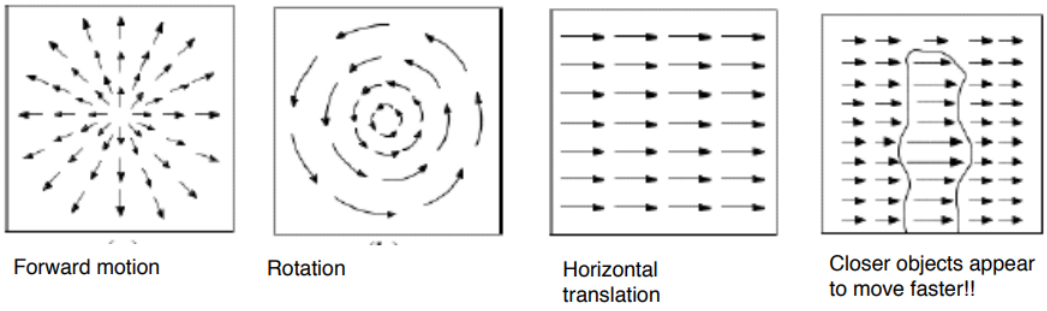


Figure 4.3: Motion fields types. Figure extracted form [2].

Dividing the motion field in small areas gives us the option to exclude areas where the motion field detected is not in accordance with the rest, such as in cases where marine animals or texture-less object are passing in the camera field of view. Moreover, for each region of the image it is

computed a vector that represents the overall movement of that region. This vector can be further used to compute the translation velocity, as described in section 4.4 as well the angular velocity, as described in section 4.5.

It is possible to divide the motion field in any number of ways, for this case it was divided in a 4 by 4 grid. By empirical experimentation, this way of dividing has a good balance between computational effort and results.

It were studied two ways of computing the average optical flow value for each region. The first was the mean value, the other one was the median value. The differences between both are presented in the subsection 4.2.1. In the figure 4.4, is represented the overall scheme followed to compute the average motion for each region of the motion field. The magnitude of the flow vector, $||(u,v)|| = \sqrt{u^2+v^2}$, is compared to a threshold, which decides what values to use to compute the average flow. The reason behind comparing the magnitude of the motion vector with a threshold is to discard values that are very small and can influence the result in a negative way. For example, if a texture-less object passes in front of the camera, the optical flow vectors, for that object, in that region, will be zero or close to zero, leading to a mean value lower than the real value. Nevertheless, not taking into consideration the small values will prevent this from happening. Each value that is higher than the threshold is then used to compute the motion vector that better represents that region.

4.2.1 Motion averaging

To compute the average value for the region of the optical flow field we can use the mean or the average. Below are presented the advantages and disadvantages of each one.

4.2.1.1 Mean value

The mean value is computed by the expression:

$$\mu = \frac{1}{N}\sum_{i=1}^{N} A_i \tag{4.2}$$

Where N in the number of motion vectors and A_i is the motion value in the u and v direction.

The problem in using the mean value is its low robustness to low and high values. This means that when computing the mean value, the same weight is given to the small and high values, in the motion field, and this can lead to a erroneous result. Even though, its computation is simple and easy to implement efficiently.

4.2.1.2 Median value

The median corresponds to the value that is in the middle of a distribution of values ordered. If the distribution has a even number of values, then the median is the mean value between the two values in the middle. If the distribution has an odd number of values, then the median values is the one in the middle. The use of the median is more robust because it has a break down point at 50%,

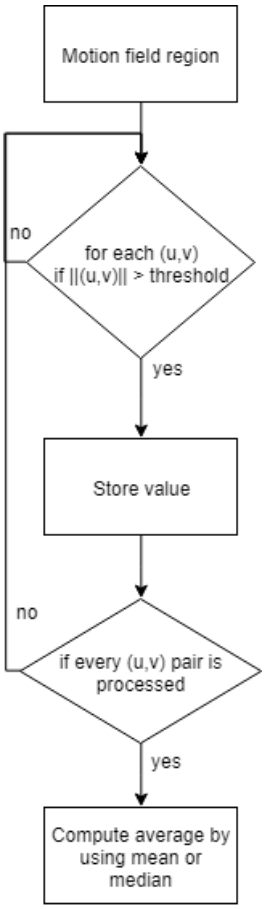


Figure 4.4: Flow chart describing the computation of the motion average value in one region of the motion field.

meaning that as long as half the values in the data are not wrong, the median will yield the right result. However, computing the median value is more computationally complex and expensive.

4.3 Temporal filtering and velocity conversion

As previously described, the slow dynamics of an AUV limit the rate of change of its speed and direction. As result, it was chosen to apply a low pass filter to the motion vectors from each one of the motion regions. This low pass filter is a simple moving average along the time domain. This reduces fast changes in direction and speed due to vibrations and small unexpected movements. The size of the moving average, unless specified otherwise, was chosen empirically as five frames, although we can change it as the navigation conditions change. Also, it is at this stage that equation 4.1 is used and the motion vectors of each region become velocity vectors. As the time between the frames is very small, it is possible to assume these velocity vectors as instantaneous velocity vectors.

4.4 Translation velocity estimation

The velocity in the x and y direction, is the translational velocity. Having the motion field reduced to 16 velocity vectors, from the 4 by 4 grid, the vector that represents the velocity of the AUV is the mean value of all the 16 velocity vectors. The averaging process also eliminates the rotation of the AUV in the z axis.

After computing the vector that represents the translational velocity in pixels per second, we translated that velocity to meters per second in the AUV reference frame. To carry out this operation it was assumed that the camera being used is a pin hole camera, with a rectilinear lens, as represented in the figure 4.5.

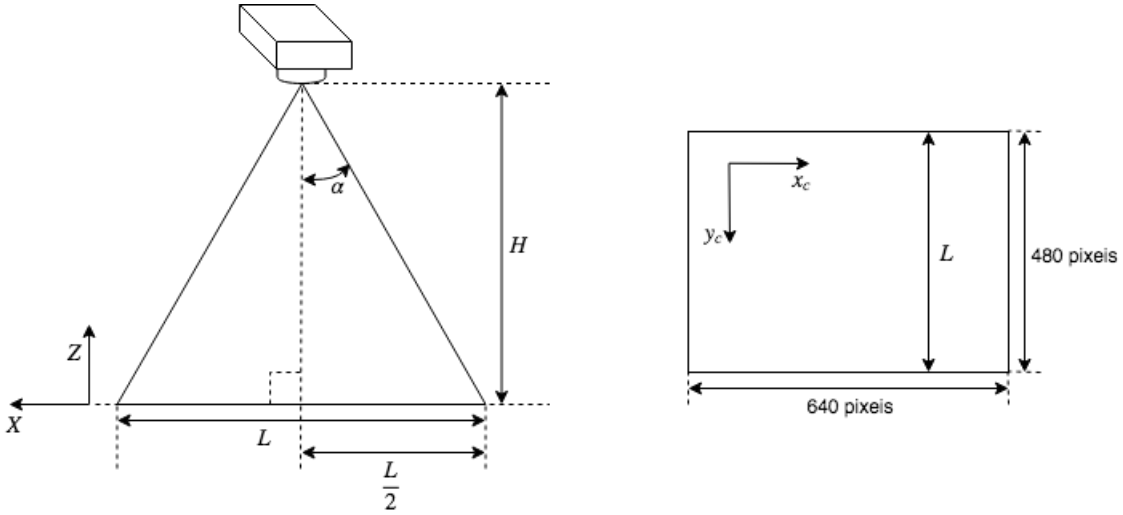


Figure 4.5: On the left is the cross section representation of the camera field of view. The α angle is half the angle of view of the camera. H is the distance of the camera to the object, which is been used to compute the velocity. L is the real size of the field of view in the x direction of the camera. The size of the frame in pixels is represented on the right.

Knowing the value of L and H, in the figure 4.5, it is possible to compute the value of α:

$$\alpha = \arctan \frac{\frac{L}{2}}{H} \tag{4.3}$$

Now, with the α angle it is possible to compute the value of L, lets call it $L(t)$ because now it can change with the $H(t)$ value, for any value of H, here $H(t)$:

$$L(t) = 2\tan(\alpha)H(t) \tag{4.4}$$

Having the $L(t)$ we can make a direct correspondence between a pixel length and the length in the world reference frame. In this case L corresponds to the x direction which has 480 pixels in size.

$$ConversionFactor = \frac{L(t)}{480}\text{m/pixel} \tag{4.5}$$

Now, we just need to multiply the velocity vector in pixels per second for the *ConversionFactor* and the velocity in pixels is converted to metres and independent of the pixels velocity and camera. Observing the figure 4.1 it is possible to convert this velocity, that is the camera frame, to the vehicle reference frame by:

$$x = y_c \tag{4.6}$$

$$y = x_c \tag{4.7}$$

4.4.1 Experimental Results

The results presented herein were computed using the description presented in section before. To compute the motion field was used the iterative and pyramidal implementation of the Lucas and Kanade, with 4 levels of the pyramid, and 2 iterations at each level of the pyramid. Also, the neighbourhood size was 11 by 11. In the first stages of the velocity estimation, division and spatial averaging, the motion field was divided in 16 equal regions, and for each one it was computed the median motion vector, using a threshold of one pixel, meaning that only motions that are higher than one pixel will be considered. Then, for the temporal filtering it was used a moving average of the 5 most recent values. A video with a VGA resolution of size 640x480 pixels was used in these experiments. To compute the α angle, at the beginning, the values of H and L were measured. The $H(t)$ value for this case is kept constant throughout the test.

For the camera used, at an height, H, of 1.59 m, the L value is 1.2 m. The α angle can then be computed by equation 4.3:

$$\alpha = \arctan \frac{\frac{1.2}{2}}{1.59} = 20.67^\circ \tag{4.8}$$

As said previously the $H(t)$ value is kept constant. In this case $H(t) = 1.44\,\text{m}$.

Using the equation 4.5, the conversion factor is:

$$ConversionFactor = \frac{2\tan(20.67^\circ)1.44}{480} = 0.0023\,\text{m/pixel} \tag{4.9}$$

With this value we can compute the translational velocity along the x and y directions. The results for a simple test sequence of the vehicle standing still and then accelerating until it reaches a constant velocity are presented in figures 4.6 (frames at instants 1 s and 5 s, with the estimated velocity vectors), are presented in the figure 4.7.

The experimental test setup for this preliminary study was build with a video camcorder pointing down from the roof of a car, figure 4.18, as this method is more simple and this way we have more observability and control over the velocity and movement of the camera. Even though the AUV is not being used nor the image acquisition is performed under water the test conditions are not significantly different. The terminal velocity of the car was measured to be at $1.81\,\text{m}\,\text{s}^{-1}$. In the figure 4.7, it is observed that the scheme presented for computing the translation velocity

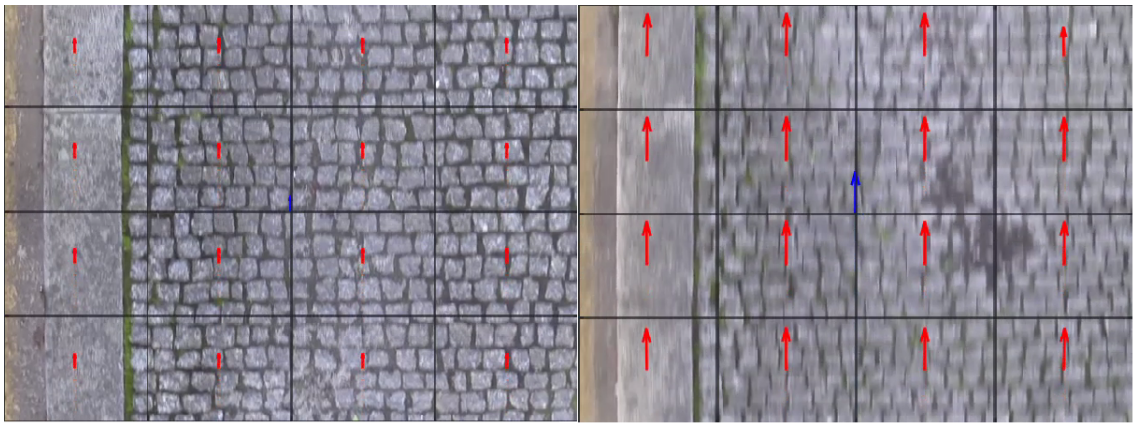


Figure 4.6: Frames captured with the velocity vectors for each region (red) and for the translational velocity (blue). The velocity vectors are in pixels per second and scaled down by 15 times. (a) is the frame at t = 1 s and (b) is the frame at t = 5 s

works with residual error. The car is only moving forward, meaning that the velocity should have just the x component converging to 1.81 $\mathrm{m\,s^{-1}}$, which is what happens in the figure. In the graphic of the y velocity we see some noise due to the car vibrations and the floor irregularity.

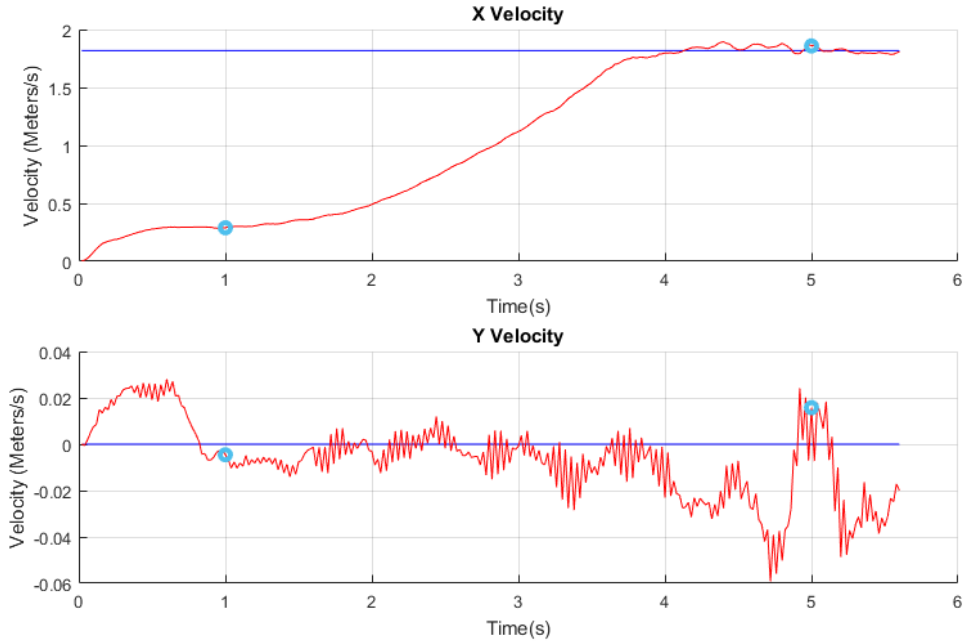


Figure 4.7: Results of the velocity estimation, in the x and y direction, for test sequence with a vehicle accelerating until it reaches 1.81 $\mathrm{m\,s^{-1}}$. The light blue dots correspond to the frames of the figure 4.6

The test described above was performed with 4 levels of the pyramid and a neighbourhood of size 11 by 11. If it was used a pyramid only with 3 levels the process fails, as the speed is too high.

This can be seen in figure 4.8, where the speed estimation does not match the true expected speed. On the other hand, the result obtained for the y velocity is very similar to the results obtained with 4 levels because the y speed is much lower.

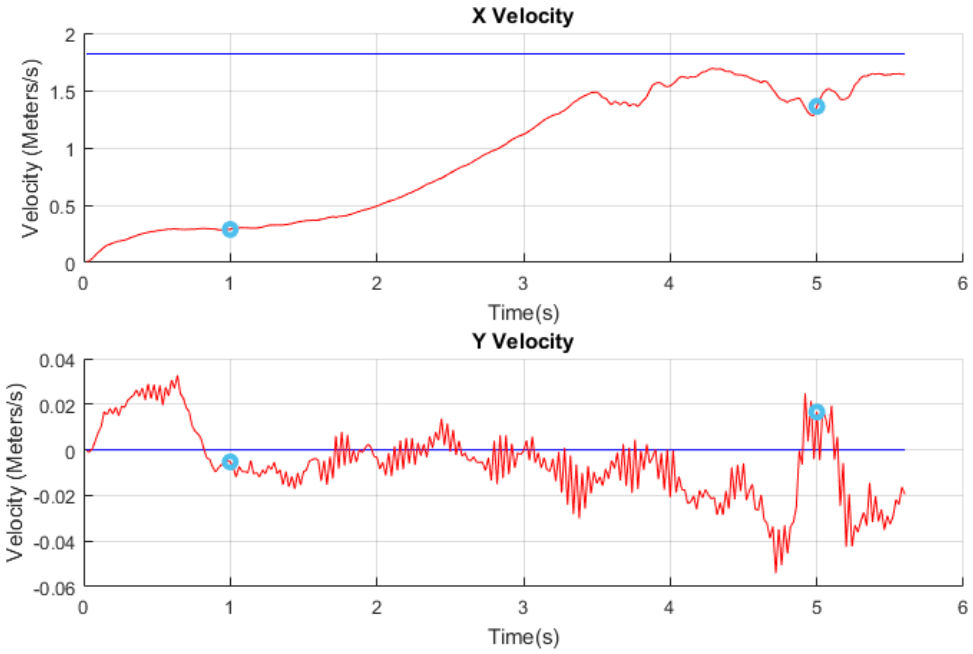


Figure 4.8: Results of the velocity estimation, in the x and y direction, for test sequence with a vehicle accelerating until it reaches 1.81 m/s. In this case using just 3 levels of the pyramid.

To counteract this phenomenon we can increase the size of the neighbourhood or the levels of the pyramid. Although, as discussed before, the better solution is to increase the levels of the pyramid.

4.5 Angular velocity estimation

Angular velocity of the vehicle is the ω velocity in the z axis, as represented in the figure 4.1. As previously explained the motion field was divided in regions and for each region we have a velocity vector that represents the average velocity of that region. Then, to reduce the noise, it was performed a temporal filtering. If we look at the motion pattern when the AUV is rotating, and if we ignore maritime structures that can move and deform, it is acceptable to assume a rigid body rotation meaning that the rotation can be expressed as a planar transformation. In this case, the body is the reference structure to where the velocity is computed relative to. Another detail to consider is that the velocity vectors of each region are in the camera reference frame, x_c, y_c and z_c, thus, after computing the angular velocity, we have to convert it to the reference frame of the vehicle. This can be performed by inverting the angular velocity at the end, i.e. $\omega = -\omega_c$.

In a planar transformation of a rigid body, the angular velocity can be computed as described in the flow diagram of the figure 4.9.

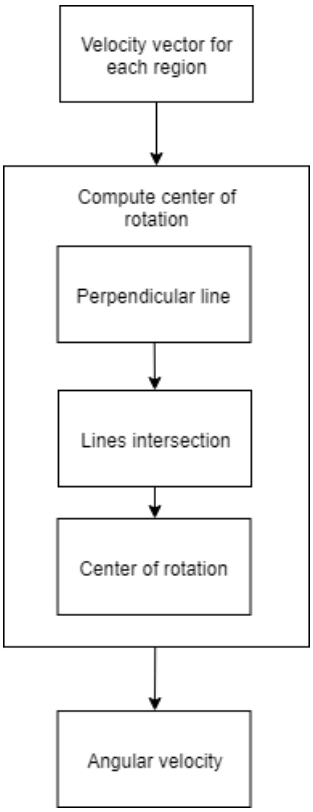


Figure 4.9: Flow diagram of the steps taken to compute the angular velocity of the AUV.

The first step to compute the angular velocity is to determine the centre of rotation, i.e. the instant centre of a planar displacement. This involves computing the perpendicular line to each motion vector, the doted line in the figure 4.10, and then computing the intersection of each line with all the others lines. The result of the intersection of the line, in the ideal case, is a unique point that corresponds to the centre of rotation, represented by the red dot in figure 4.10. However, as the velocity vectors are not ideal, it will appear a cloud of intersection points in the position of the centre of rotation. The centre of rotation can be estimated by computing the centre of the cloud of points. After, we can compute the angular velocity, by knowing the distance between the centre of rotation and the origin of the velocity vectors, and computing the average of all angular velocities. In the next sections it will be presented a detailed explanation of each step.

4.5.1 Computing the perpendicular line

The perpendicular line, doted line in the figure 4.11, is the perpendicular line to the vector, (u,v), that passes in the point (X_0,y_0). In the figure 4.10, it is also possible to observe all the perpendicular lines of the frame and their intersection.

The line can be defined by its linear equation:

$$y = mx + b \tag{4.10}$$

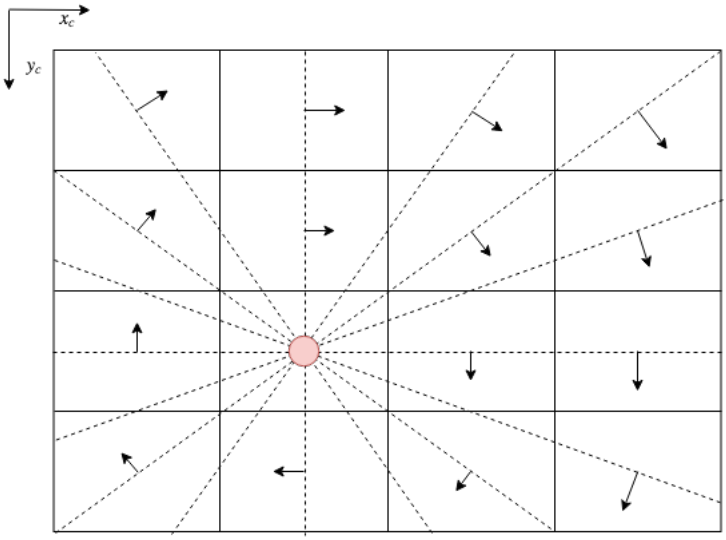


Figure 4.10: The red dot is the centre of rotation which coincides with the intersection of the perpendicular lines to the velocity vectors of each region.

The m and b values are estimated as presented next:

The slope, m, can be computed by rotating 90 degrees the vector (u,v) and then computing its slope:

$$m = \frac{-u}{v} \tag{4.11}$$

Note that in case the velocity in the v direction is 0 the slope will be infinite.

Afterwards, it is computed the b value. In order to calculate it we need to know the value of (x_0,y_0) and (x_1,y_1). Being (x_0,y_0) the position of the beginning of each velocity vector, in this case it corresponds to the centre of each region:

$$x_0 = \frac{secWidth}{2} + secWidth \times j \tag{4.12}$$

$$y_0 = \frac{secHeight}{2} + secHeight \times i \tag{4.13}$$

where $secHeight$ and $secWidth$ is the height and width of each region, respectively, and, j and i are the index of each region, starting at $(0,0)$. (x_1,y_1) can be computed by:

$$x_1 = x_0 + u \tag{4.14}$$

$$y_1 = y_0 + v \tag{4.15}$$

Next, the b is computed by:

$$b = y_0 - m \times x_0 \tag{4.16}$$

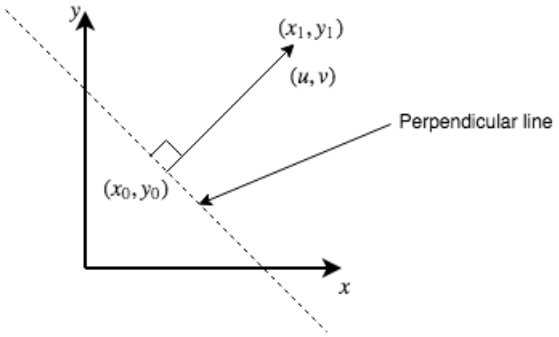


Figure 4.11: The doted line is the perpendicular of the vector (u,v) that passes in the point (x_0, y_0).

In case of m being infinite the line is discarded. This will not affect our computation because the probability of appearing a vertical line is very low, and even more low is the probability that there is a rotation and most of the lines are vertical. Moreover, when the movement is pure translational in the y direction, it is expected that all lines are vertical, though in this case the angular velocity is zero.

4.5.2 Computing the lines intersection

As described before we need to compute the point of intersection of each line with all the others lines. As the intersection of the line A with line B is equal to the intersection of B with A, we just need to compute the intersection as in the table 4.1.

Table 4.1: Points of lines intersection.

		i			
		1	2	...	n
j	1	$P_{1,1}$			
	2	$P_{2,1}$	$P_{2,2}$		
	...	$P_{j,1}$	$P_{j,2}$	$P_{j,i}$	
	n	$P_{n,1}$	$P_{n,2}$	$P_{n,i}$	$P_{n,n}$

The computation of the point of intersection (x,y) for two generic lines can be preformed by:

$$m_1 \times x + b_1 = m_2 \times x + b_2 \tag{4.17}$$

solving for x:

$$m_2 \times x - m_1 \times x = b_1 - b_2 \tag{4.18}$$

$$x(m_2 - m_1) = b_1 - b_2 \tag{4.19}$$

$$x = \frac{b_1 - b_2}{m_2 - m_1} \tag{4.20}$$

Then the y value can be computed by:

$$y = m_1 \times x + b_1 \tag{4.21}$$

Note that in this case we could use m_2 and b_2 that would make no difference.

In equation 4.20, if m_1 and m_2 are equal it will mean that the two lines are parallel, thus they do not intersect. In this case, we consider the point of intersection of the two lines will be at infinite and as result the point is discarded for the further steps.

4.5.3 Estimating the centre of rotation

In the figure 4.12 are presented the steps taken to compute the centre of planar displacement, or in this case, the centre of rotation. Each step is explained below.

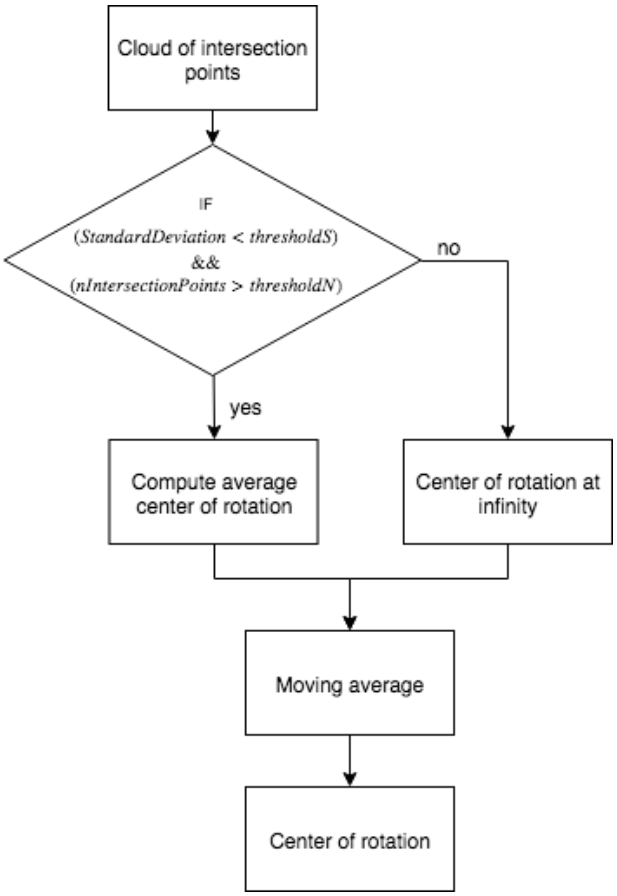


Figure 4.12: Flow diagram with the steps to compute the centre of rotation.

After we have the set of intersection points, we can compute the mean centre of rotation as the representative centre of rotation of the AUV. Still, we have to decide when the AUV is really rotating or when it is just noise, and the AUV is in reality in a translation movement. When the AUV is making a translational movement, in a real scenario, the motion vectors will not be perfectly parallel to each other making the perpendicular lines to intersect. Moreover, if we

consider the AUV rotating slowly but with a high translational velocity, the main component in the motion vectors will be the translational one. This will originate perpendicular lines that will be almost parallel and the cloud of the intersection points will be very disperse, with high variance. Also, the number of intersection points that were considered to be at infinite will be high, meaning a low number of intersection points. Thus, we need a way to decide when we have reliable information to compute the angular velocity and when we do not.

One way to decide if we have enough information to compute reliably the centre of rotation is having a sufficient number of valid points, as a low number of valid points means that the movement is mostly translational and most of the points were discarded. Adding to this, we can compute the standard deviation of the set of intersection points. If they are close to each other, the standard deviation will be low, which means that the estimate is more probably to be correct. On the contrary, if the points are further apart, the standard deviation will be higher, meaning that there is a lower probability of the estimation being correct. Defining the maximum standard deviation and the minimum number of points necessary for the computation is a difficult task and depends on the application and the trade-off between computing the angular velocity in any occasion and not computing the velocity unless the AUV is explicitly rotating.

At last, to compute the centre of rotation, when decided that is reliable enough, it can be used the mean value or the median value. The trade-off between the two are explained in 4.2.1. After the instantaneous centre of rotation for a frame is computed, it is performed a moving average. The purpose of computing the moving average is to smooth the movement of the centre of rotation from one frame to another, as it is not expected that it changes place very quickly. Furthermore, the moving average is computed only with the centres of rotation considered valid . In case that all the values to compute the centre of rotation are at infinity then it is assumed that the AUV is most probably translating and not rotating.

4.5.4 Estimating the angular velocity

After we have computed, in the previous steps, the centre of rotation, point (x_C, y_C) in figure 4.13 as well V_1 and V_2, which are two generic velocity vectors of each region, we can compute the angular velocity, ω, which is the angular velocity of the AUV, as described next:

Note that V_1 and V_2, in figure 4.13, are perpendicular to the line that intercepts: the points of vectors origin, (x_1, y_1) and (x_2, y_2); and the centre of rotation, (x_C, y_C). r_1 and r_2 are the distance between the points (x_i, y_i) and (x_C, y_C).

As V_i is perpendicular to r_i, the angular velocity ω can be computed by:

$$\omega = \frac{r_i}{V_i} \tag{4.22}$$

But first we need to compute r_i and V_i. V_i is the magnitude of the velocity vector of the region and can be computed by:

$$\begin{aligned} V_i &= ||(u_i, v_i)|| \\ V_i &= \sqrt{u_i^2 + v_i^2} \end{aligned} \tag{4.23}$$

To compute r_i, first we need to compute (x_i, y_i), which can be done by the equations:

$$x_i = \frac{secWidth}{2} + secWidth \times j \tag{4.24}$$

$$y_i = \frac{secHeight}{2} + secHeight \times n \tag{4.25}$$

Where j and n are the indices of the motion region. Next the r_i can be computed by:

$$r_i = \sqrt{(x_C - x_i)^2 + (y_C - y_i)^2} \tag{4.26}$$

Then, from equation 4.22, 4.24 and 4.26 we have:

$$\omega = \sqrt{\frac{(x_C - x_i)^2 + (y_C - y_i)^2}{u_i^2 + v_i^2}} \tag{4.27}$$

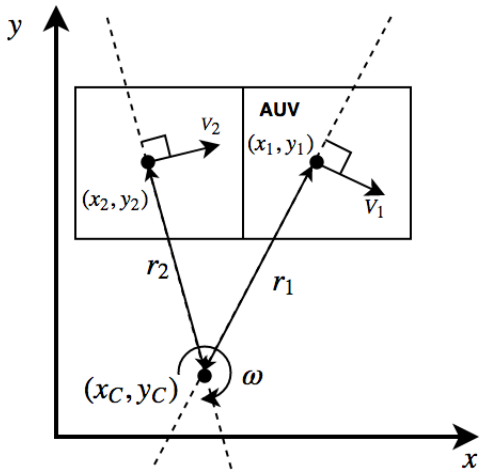


Figure 4.13: Diagram of the angular velocity estimation.

As we have multiple vectors V_i we have to compute the angular velocity ω for every velocity vector of each region. Then, the final angular velocity is the mean value of all the angular velocities.

Although, we are still missing the signal of the angular velocity. This signal can be computed by the cross product of the r_i by V_i.

$$s = r_i \times V_i \tag{4.28}$$

As the the two vectors are two dimensional, the z component will be 0. Then the cross product can be computed using its matrix notation and then computing the determinant of the 3 by 3 matrix, resulting in:

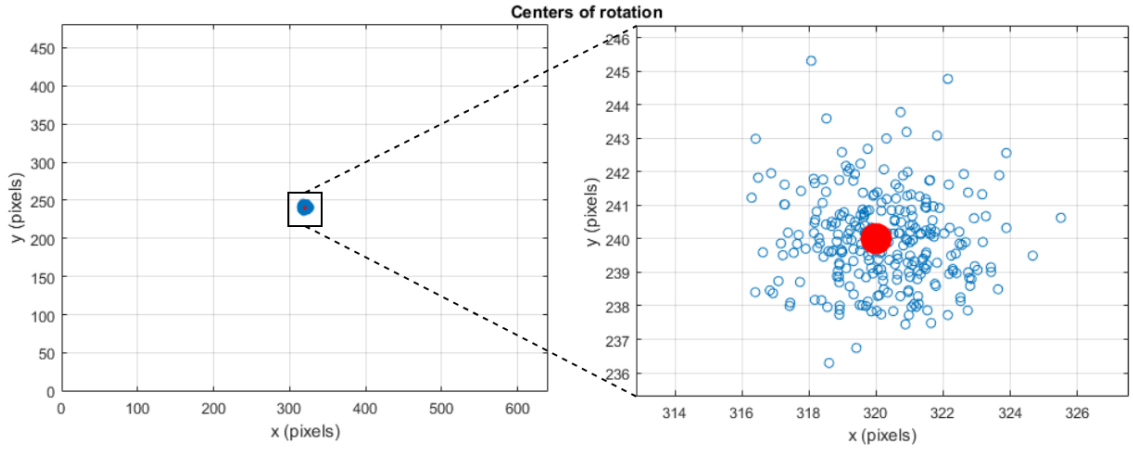
$$s = (x_C - x_i)v_i - (y_C - y_i)u_i \tag{4.29}$$

The s value is computed for every velocity vector of each region. Then, we can verify by the sum of the all s values if the angular velocity is positive, then the AUV is rotating counter clockwise, or negative, meaning that the AUV is rotating clockwise.

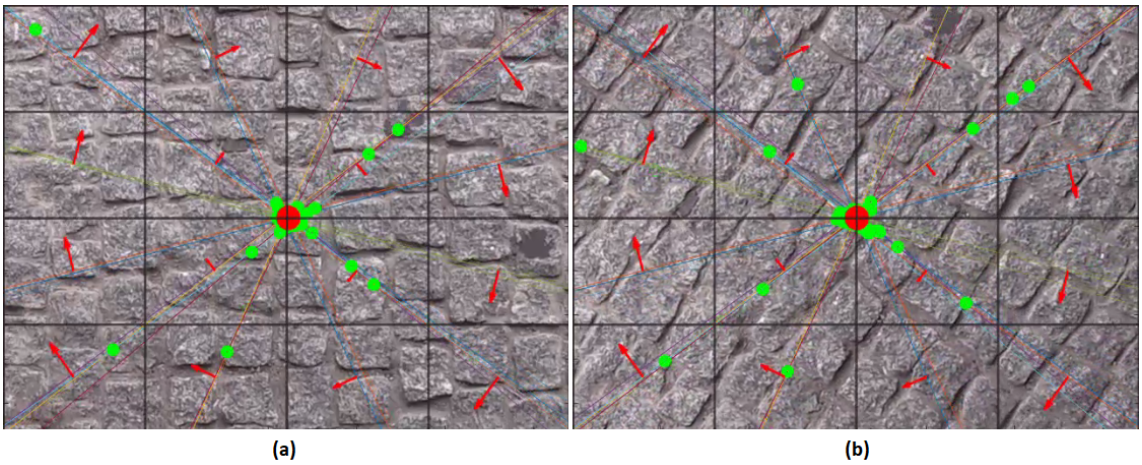
4.5.5 Results

Similarly to the estimation of the translational velocity, the results for angular velocity determination were computed using the iterative and pyramidal implementation of the Lucas and Kanade, with 3 levels of the pyramid, instead of 4 as used before (in this case the pixels motion is smaller and 3 levels are sufficient to compute it correctly), and 2 iteration at each level of the pyramid. The neighbourhood size was 11 by 11. In the first stages of the velocity estimation, the motion field was divided in 16 equal regions, and for each one was computed the median motion vector, using a threshold of one pixel, meaning that only motions that are higher than one pixel will be considered. Then, for the temporal filtering was used a moving average of the 5 most recent values. In this test was used a video with VGA resolution of size 640x480 pixels. The video sequences used in these tests were originated using Matlab.

The first video sequence is a camera rotating 90° clockwise, with the centre of rotation in the middle of the frame, figure 4.14b. This sequence simulates the AUV rotating with the centre of rotation right in the middle of the camera frame. The $-90°$ rotation is done with an angular velocity of $-10\,°\,s^{-1}$. Note that to compute the angular velocity we do not need to know the distance to the object.



(a) Graphic with the different centres of rotation computed in each 270 frames of the video. Left graphic is all the image frame and the right graphic is the zoom in on the centre



(b) Frames of the process to compute the angular velocity.

Figure 4.14: Figures describing the process of computing the angular velocity. The red vectors are the velocity vectors, in pixels per second, of each region, and point in the direction of the camera velocity. The green dots are the intersection points of the perpendicular lines, the big red dot, at the centre, is the computed centre of rotation, and the blue dots are the computed centre of rotation for each video frame.

The angular velocity is in the graphic of the figure 4.15. The blue line is the true angular velocity of the camera, which is negative because the camera is rotating clockwise. The red line is the angular velocity computed by the algorithm that coincides with the true angular velocity. Moreover, in the top right corner of the figure is the total angular displacement of the camera, that should be $-90°$. By comparing the true displacement and the computed displacement, as well the velocities, we conclude that the algorithm performs as expected, with a relative error inferior to 0.1%. Also, in the 4.14a, we can observe that the centre of rotation computed for every frame of the video is very close to the real centre of rotation (red dot).

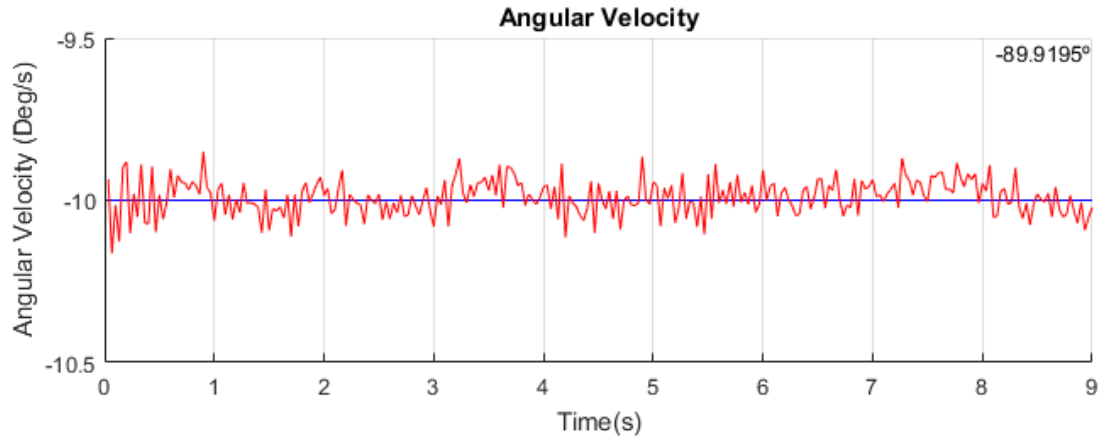
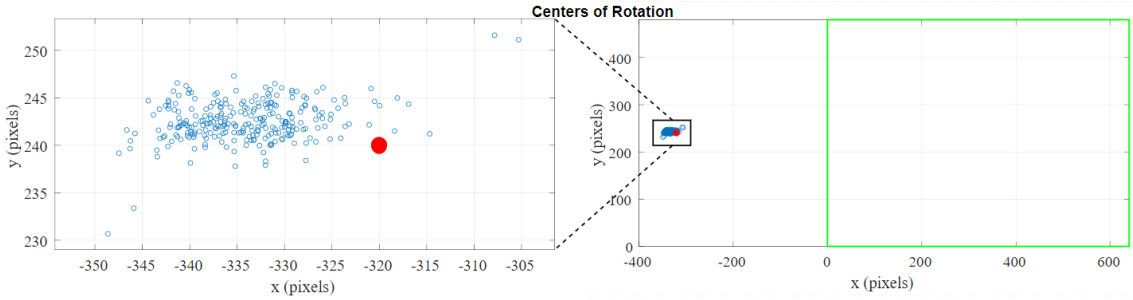
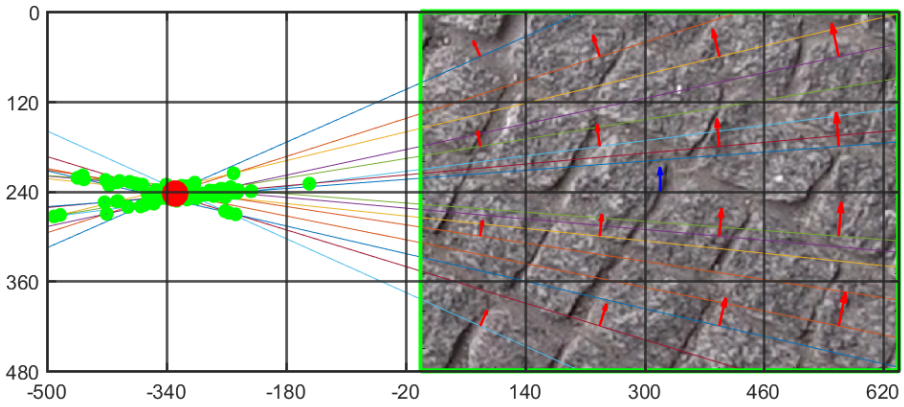


Figure 4.15: The blue line at $-10°$ is the correct angular velocity. The red line is the computed angular velocity. In the top right is the total angular displacement.

In the next test, it was used the same texture as in the figure 4.14b, but, this time the centre of rotation was out of the frame. The camera still rotated 90°, counter clockwise, at a velocity of $10\,^{\circ}\,s^{-1}$.



(a) Graphic with the different centres of rotation computed in each frame of the video



(b) Frame of the process to compute the angular velocity.

Figure 4.16: Computing the angular velocity with the centre of rotation out of the camera angle of vision.

As shown in the figure 4.16, the centres of the rotation, computed for each frame, are outside the frame around the position $(-335, 243)$. However, the red dot specifies where the real centre of rotation is. This means that, as the centre of rotation is further way from the centre of the frame, the worst will be our estimative of the angular velocity.

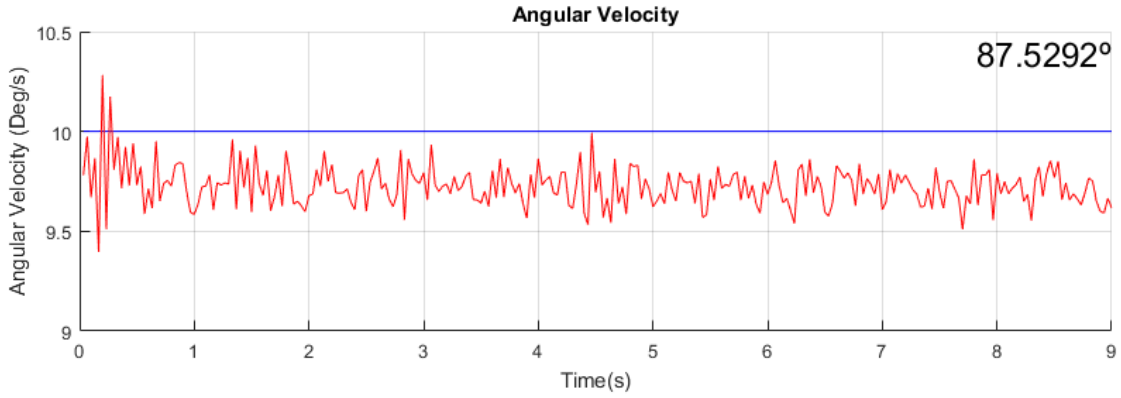


Figure 4.17: Angular velocity computed with the centre of rotation out of the camera field of view. The blue line at $10\,^{\circ}\,s^{-1}$, is the true angular velocity of the camera, the red line is the computed angular velocity.

The figure 4.17 shows that the error of the estimative of the angular velocity has increased to 2.74% that corresponds to what was said before. As the centre of rotation starts to be wrongly estimated it influences the angular velocity estimative.

This error begins to appear because, if we keep the angular velocity constant and move away the centre of rotation from the centre of the frame, the higher will be the velocity of the pixels in the frame. As a consequence, the optical flow accuracy is reduced starting to appear more noise, thus increasing the error of the angular velocity for higher translational velocities. Moreover, as said previously, we assume that the velocity vectors are instantaneous although, this approximation is only valid for small displacement between frames. As the velocity increases this approximation starts to introduce a larger error.

4.6 Experimental results and comparison between optical flow methods

In this section, it is presented the final results and a comparison between the two optical flow methods, iterative and pyramidal Lucas and Kanade and block matching, presented in the chapter 3. To test which one of these methods has better results for our final application, both methods were subject to some tests, with a test setup created to simulate some of the moving dynamic of the AUV. The tests that were performed had the objective to evaluate the ability to compute the angular and translational velocity from the results of each one of the methods.

The test setup is illustrated in the figure 4.18. As previously explained, it was used a camera mounted on a car. The camera is facing down and its angle relative to the perpendicular, β can be changed. The camera is at an altitude from the ground, $H(t)$, that can vary with the changes in the ground. The initial height of the camera, $H(t)$, is 1.59 m , and it is kept constant except for changes in the ground. The angle β is 0° unless it is said otherwise.

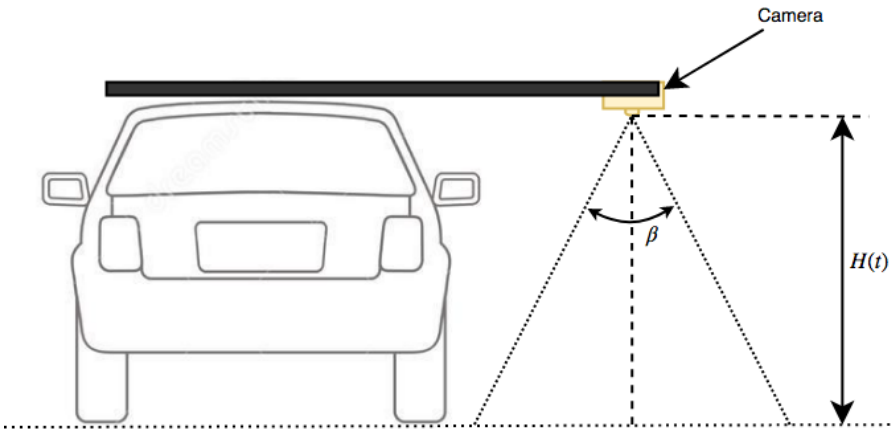


Figure 4.18: Back view of the test setup used.

First are presented the results for computing the translational velocity in section 4.6.1, then are presented tests to the angular velocity computation in 4.6.2.

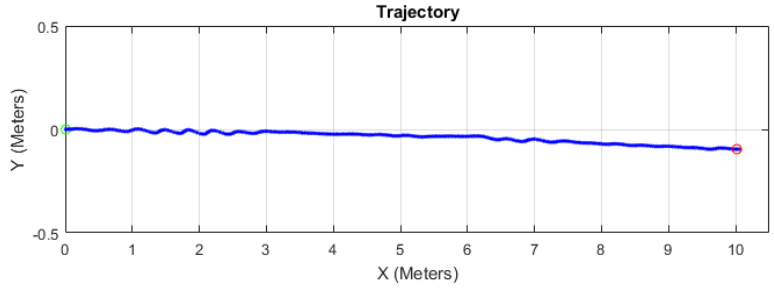
4.6.1 Translational velocity

To test how the algorithms performed computing the translational velocity, it was performed a simple test using a vehicle running in a straight line at a velocity of 1.81 $\mathrm{m\,s^{-1}}$ during proximately 10 m.

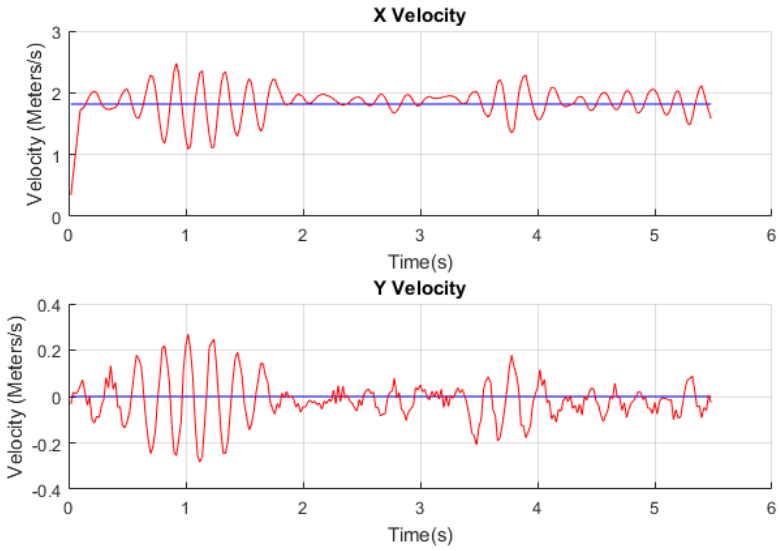
For the block matching method, the frame was divided in 4 by 4 macro blocks. It is not possible to use a higher division because we have to use a multiple of 4 so we can divide in equal parts. Therefore, the next possible division would be 8 by 8. However, such high division will make the search windows so small that, with this velocity, the algorithm will fail. For the Lucas and Kanade it is used a pyramid with 3 levels, 2 iterations at each level, and a 11 by 11 window.

In the figure 4.19a we can observe that the trajectory is mostly correct. The green dot is the starting point and the red dot is the ending point. As said before, the car travelled a distance of 10 m in the x direction and 0 m in the y direction thus, in the end the the distance computed is 10.05 m in the x direction and 0.1 m in the y direction resulting in a relative error of 0.5%. In the figure 4.19b are represented the components of the velocity of the vehicle. We can observe a lot of oscillations in this components due to the vibration of the camera. Nevertheless, it is visible that the velocity computed is close to the real velocity, represented in blue. Looking at these results we conclude that the algorithm can compute correctly the velocity of the vehicle up until velocities of 1.81 $\mathrm{m\,s^{-1}}$, at a distance of 1.59 m and at 50 FPS. Although, when the optical flow velocity is

higher, as in the figure 4.8, we have to add another level to the pyramid, becoming a pyramid with 4 levels. To overcome this problem the algorithm must have the ability to change the number of pyramid levels with the cost of increasing the computing complexity.



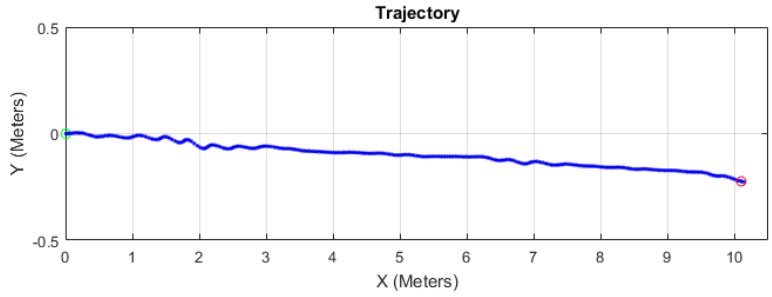
(a) Computed trajectory of the car



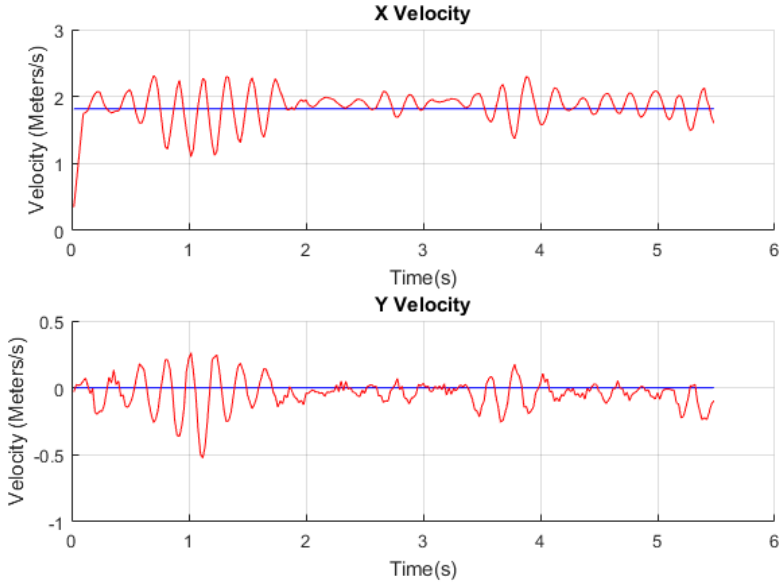
(b) Computed velocity of the car in the x and y direction (red) and true velocity (blue).

Figure 4.19: Velocity and trajectory of the car computed using the iterative and pyramidal Lucas and Kanade.

For the block matching algorithm, the trajectory of the vehicle, in the figure 4.20a, is almost the same as the one computed with the Lucas and Kanade. However, it has introduced more than the double of the error in the final position of the vehicle, namely 10.13 m in the x direction and -0.23 m in the y direction, resulting in a relative error of 1.33%. Remember that the final position should be at (10 m, 0 m). Concluding, this algorithm can compute the velocity correctly, although, it is not as precise and accurate as the iterative and pyramidal Lucas and Kanade. Another down point is that the tests were performed in a controlled environment where the sequence is not very difficult. As we increase the sequence difficulty the results become worse since the algorithm is not as robust as the iterative and pyramidal Lucas and Kanade. Moreover, as previously said, we loose the small details of the moving scene, and even though we do not need them for our final application, these details can be used in some future works.



(a) Computed trajectory of the car



(b) Computed velocity of the car in the x and y direction.

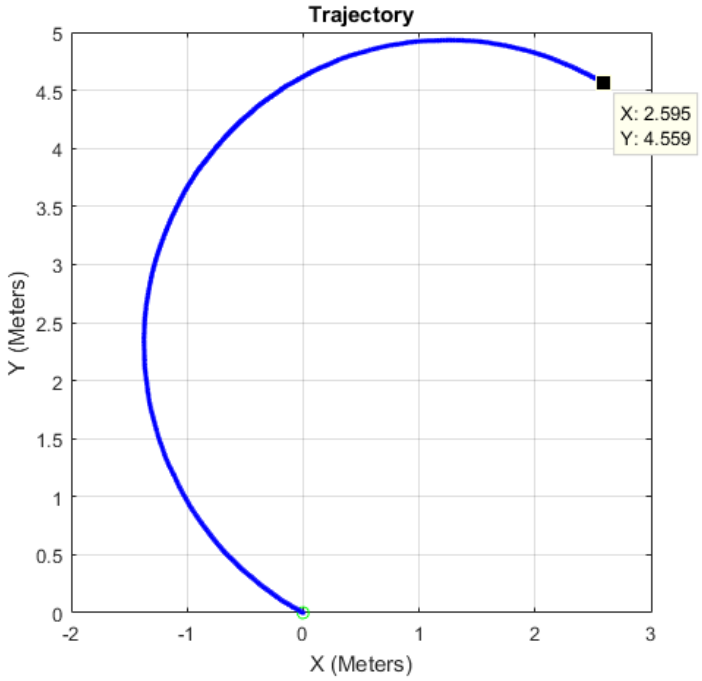
Figure 4.20: Velocity and trajectory of the car computed using block matching

4.6.2 Angular velocity

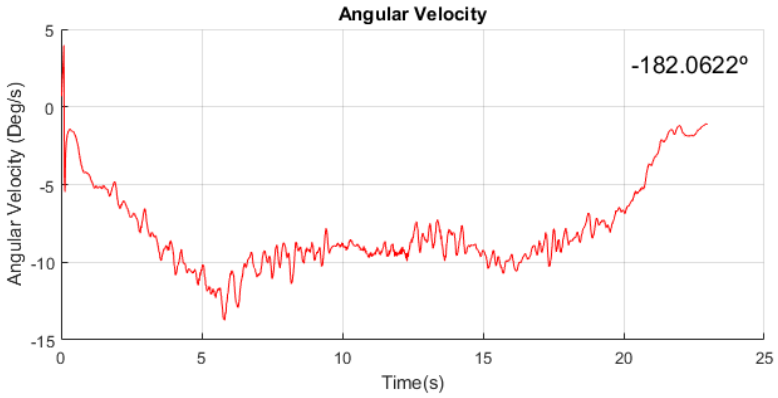
For the angular velocity test, it was performed an 180° turn with the wheels turned all the way. It was also measured the diameter of the turn as 5.10 m. In this case we are not measuring the real angular velocity at any instance, instead we will observe the final orientation computed by the algorithm and the trajectory, and compare them to the real value.

Using the same parameters for the estimation of the translation velocity, and just changing the size of the moving average to 10, so the signal has less noise, the results for the Lucas and Kanade pyramid and iterative implementation are presented in the figure 4.21. Starting in the position (0,0), the car describes a circular path ending in the position (2.59, 4.56). This circular trajectory has a diameter of 5.24 m, which is close to the 5.10 m measured on the site, resulting in a relative error of 2.75% in the diameter. Moreover, we can observe that the trajectory is very close to a circular path as it was expected. In the figure 4.21 is the angular velocity computed, with a moving average of 10 frames, to eliminate the high noise due to the vibrations of the camera. If the

angular velocity is integrated along the time, the angular displacement of the vehicle is obtained. This displacement should be -180° and the algorithm computed it as being -182°, which is very close to the real angular displacement, yielding a relative error of 1.11%.



(a) Computed trajectory of the car

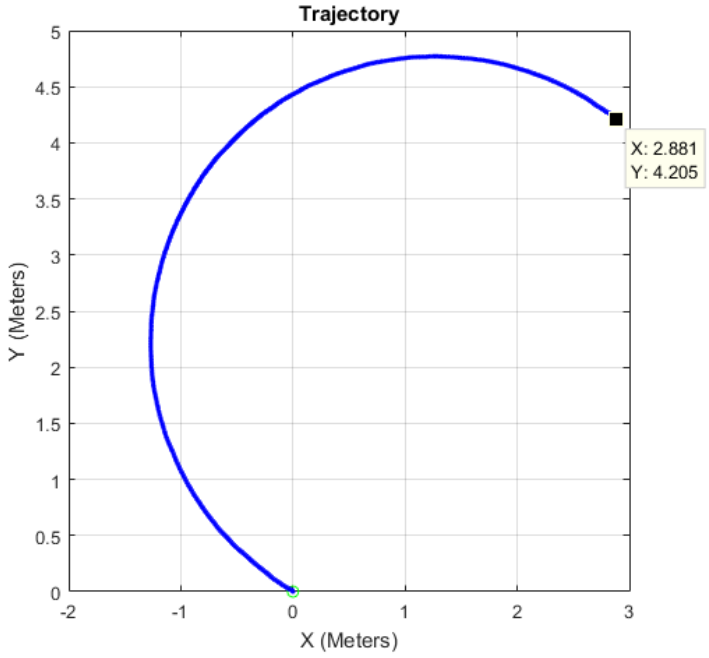


(b) Computed angular velocity of the car.

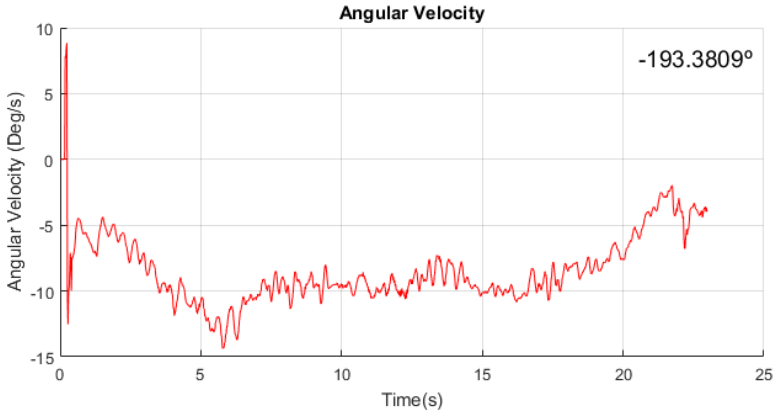
Figure 4.21: Angular velocity and trajectory of the car computed using the pyramidal and iterative Lucas and Kanade.

Regarding the block matching results, we see a decrease in the accuracy, as expected from the previous results. Using the same parameters of the Lucas and Kanade, the results for the angular velocity with this algorithm are presented in the figure 4.22. In the figure 4.22a, the final position is at the point (2.88, 4.20), meaning that the diameter of the circle is 5.09 m, which is very close to

the 5.10 m measured, with a relative error of 0.2%. However, the final angular position computed is $-193.38°$ which is over the $-180°$ that should be. The relative error is 6.9%.



(a) Computed trajectory of the car.



(b) Computed angular velocity of the car.

Figure 4.22: Velocity and trajectory of the car computed using block matching

Another test that was performed to analyse the ability of the algorithms to compute the angular velocity was to rotate the camera $-90°$, trying to maintain it in the same place, but this time with a higher difficulty pattern and a changing centre of rotation close to the centre of the frame. The comparison is presented in the figure 4.23. As we can observe, the block matching presents a bad result, with the angular velocity being mostly noise. On the other hand, although the Lucas and Kanade was not exactly accurate, it was able to give an approximate estimative of the the angular velocity. Because of the Lucas and Kanade being more robust and able to give more precise and

accurate results, it was decided that this should be the algorithm used for the future work.

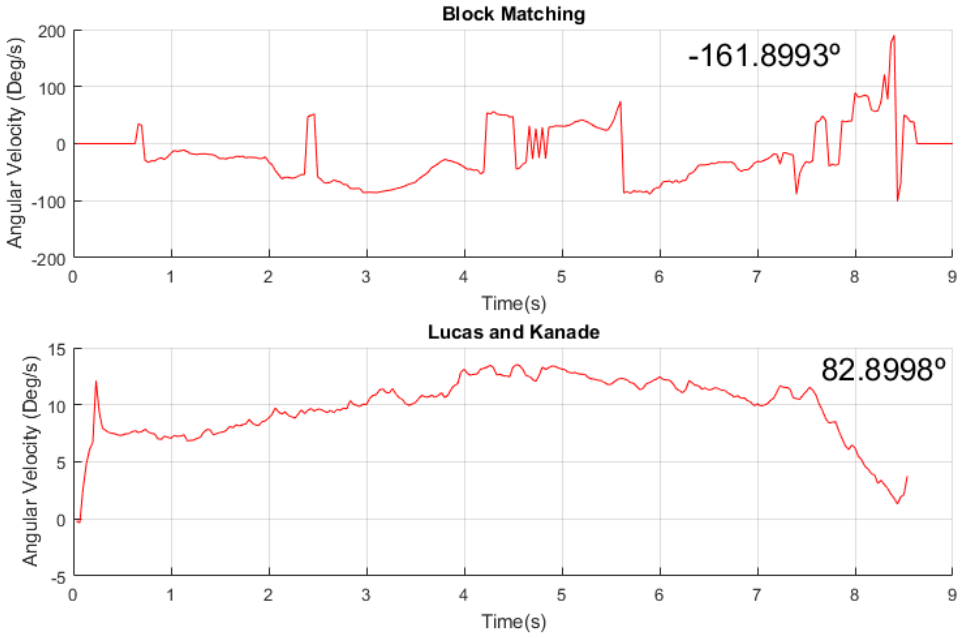


Figure 4.23: Angular velocity computed with the two different algorithms.

4.6.3 Changing the β angle

In the figure 4.24 are the results for the translational velocity of the vehicle using a β angle of 40° and using the pyramidal and iterative Lucas and Kanade. The results show that is possible to compute correctly the translational velocity of the vehicle. The $H(t)$ distance has to be adapted to the angle of the camera, if $H_0(t)$ is the height with β equal to 0°, then the $H_{40}(t)$ is the computed by:

$$H_{40}(t) = \frac{H_0(t)}{\cos(\beta)} \tag{4.30}$$

It is observable that at time 8 s, the x component of the velocity rises up, due to the parallax effect described in the section 4.1.1, where an object passes in front of the camera closer then the others objects around it. The has the distance for this object is nor corrected the perceived velocity is higher. Until approximately 3 s, the car is accelerating to the $1.81\,\mathrm{m\,s^{-1}}$.

4.6.4 Conclusions

From the results presented in the previous subsections, we conclude that it is possible to estimate the vehicle angular and translational velocity using the algorithms presented in the chapter 3, namely the iterative and pyramidal Lucas and Kanade and the block matching, followed by the algorithm presented in the chapter 4. However, the algorithms yielded different final results. The block matching presents good results for computation of the angular and translational velocity,

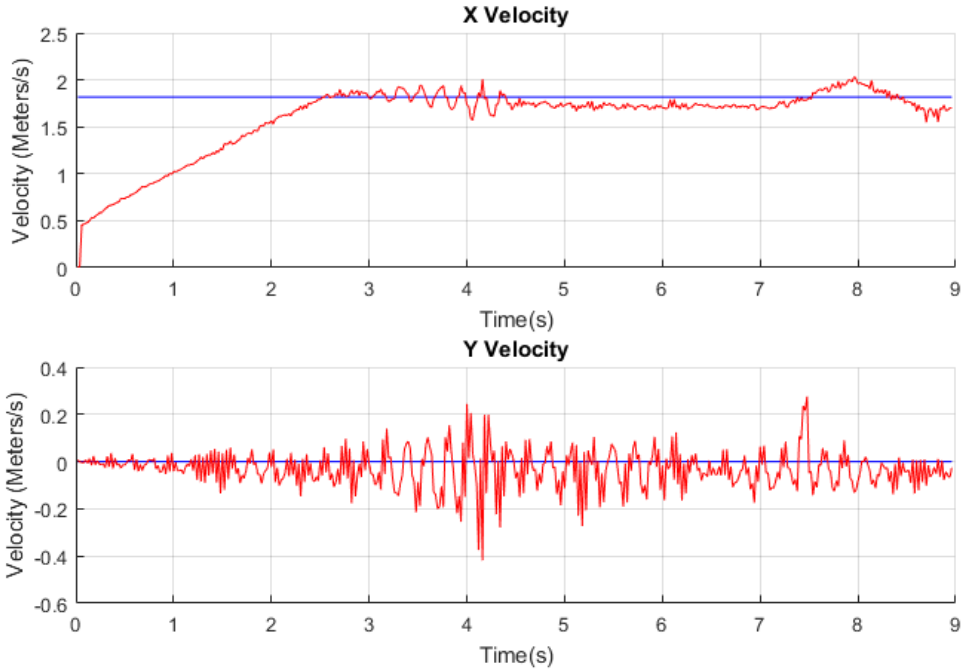


Figure 4.24: Translational velocity computed with a β angle of 40°.

with a relative errors of 1.32% for the pure translational velocity test, 0.2% for the diameter and 6.8% for the angular velocity in the angular velocity test. Nonetheless, it was observed that when the visible pattern for computing the velocity has less features, the algorithm fails. On the other hand, the iterative and pyramidal Lucas and Kanade presents better results than the block matching for the translational and angular velocity, with a relative errors of 0.5% for the pure translational velocity test, 2.74% for the diameter and 1.1% for the angular velocity in the angular velocity test. Although in some of the test the two were close, with the Lucas and Kanade having a small advantage, it was shown that the Lucas and Kanade is more robust, being able to produce better results in a scene with less patterns. For the reasons, it was decided that the iterative and pyramidal Lucas and Kanade should be the algorithm to be implemented in the final solution.

Regarding the angular velocity estimation, it was also shown that as the centre of rotation moves away from the centre of the frame, its estimation becomes less accurate. This is due to: (i) a larger dispersion of the intersection points of the perpendicular lines because of the small variations in the pattern, which introduce errors in the velocity vectors that become amplified through the perpendicular line; and (ii) the assumption of instantaneous velocity vectors introduces larger errors as larger the displacement between to consecutive frames. Also, we have to take into consideration that, if a vehicle is in a translation movement with very low angular velocity, thus the centre of rotation far way from the centre of the frame, it will be very difficult to know if the vehicle is really rotating or just in a translational movement. In this cases, we need to use additional information, from other sensors in the vehicle, to compute the angular velocity.

Another important conclusion is that, although we are using the iterative and pyramidal Lucas

and Kanade because of the velocity limitation of the simpler version, this algorithm still has a limit for a maximum velocity of the pixels in function of the number of levels of the pyramid and the size of the neighbourhood. This translates in a maximum vehicle velocity that the algorithm is able to compute depending on the distance at the structure which the velocity is being relatively computed to.

At last, an important limitation of this algorithm we need to mention, and that is inherent to the computer vision systems, is the aperture problem already discussed. This problem can cause the system to compute the vehicle velocity in the wrong direction. An integration with another sensors can solve this problem by creating redundancy on the system.

Chapter 5

Hardware architecture

Following the results presented before, the next step is to develop an architecture for the computation of the iterative and pyramidal Lucas and Kanade. As it was said before, the optical flow computation is a computationally intensive task that is incompatible with low performance embedded systems for AUVs. To solve this problem, it was decided that the best solution is to implement the optical flow algorithm in an FPGA. In this chapter we propose a hardware architecture suitable to be implemented in low end FPGAs and achieve real time for the calculation of dense optical flow for VGA images, using the iterative and pyramidal Lucas and Kanade algorithm. First is presented a system overview with the main components of the system (section 5.1). Then it is discussed the memory requirements and for the system to be implemented such as the others subsystems. At the end, is presented an estimation of the system requirements.

5.1 System overview

The system overview, in the figure 5.1, can be divided in six main components:

1. **Pyramid construction and filtering** - Construction of the pyramid with subsampling stages, Gaussian smoothing stages and computation of the spatial derivatives for each level. Further details in subsection 5.3.

2. **Memory** - One of the most important parts of the system since we are working with images that require a lot of space and we want a real-time system, thus we need to have a convenient memory architecture that is fast enough, to meet the timing requirements, and space consumption. More details in subsection 5.2.

3. **Module responsible for updating the optical flow value** - The optical flow value for each pixel has to be constantly updated and, as we go through the levels, has to be resized for the next level.

4. **Controller for the system** - Responsible for controlling all the different parts of the system.

5. **Matrix creation** - Matrix creation stage for computing the optical flow (subsection 5.5).

6. **System solver** - Solves the least squares problem and validates the output. In subsection 5.6.

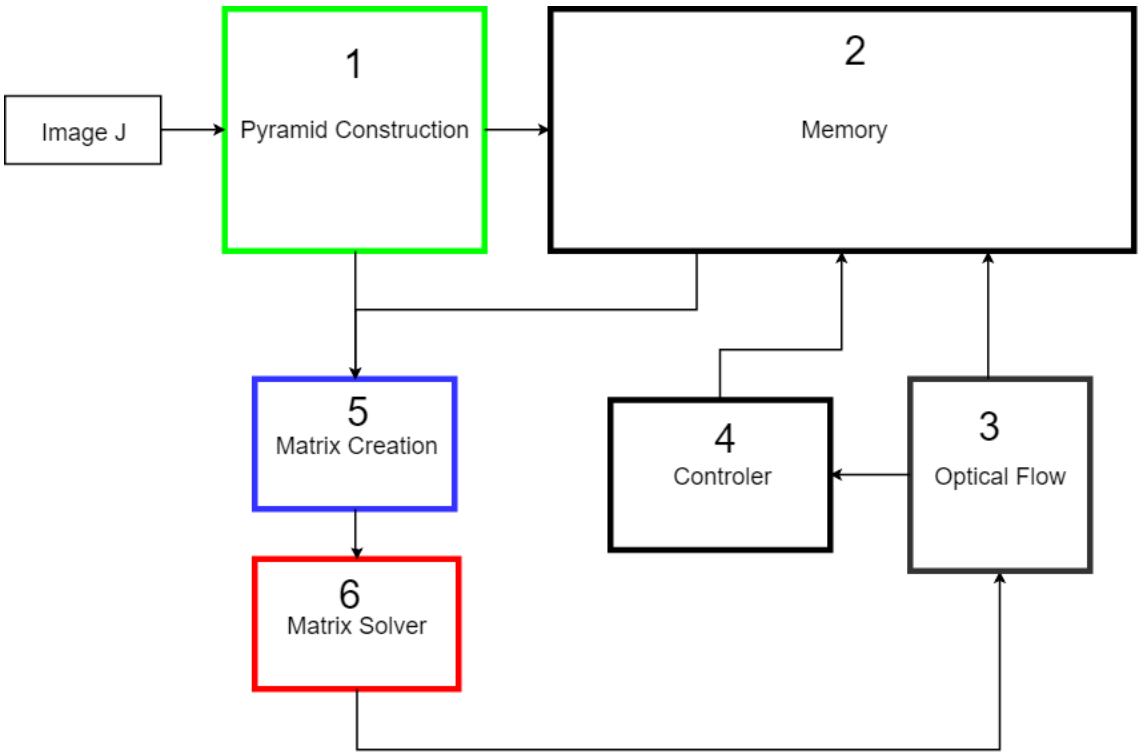


Figure 5.1: System overview of the FPGA implementation. The parts 1, 5 and 6 will be presented below.

The target FPGA for this project is the Spartan 6 and the board considered as reference target is the Atlys. As said previously the memory is on of the most important parts and the bottleneck of our system. The memory block in the figure 5.1 represents the interface with an external dynamic memory, as currently exists in several FPGA-based boards, including the Atlys board. This board has a 128 Mbyte DDR2, supporting a data rate of 400 Mbit per second and 16 data lines. The interface to the dynamic memory in implemented with the XILINX memory interface generator (MIG) tool [34, 35].

5.2 Memory

The access to the memory is critical for this application. First, we need to save the smoothed image at each level, and the x and y derivatives. Estimating the bandwidth necessary to save all this images and derivatives can be performed as follows: if we assume that we can encapsulate the derivatives and the image in one 32 bit word (note that we want to read all of them at the same time, i.e. when we want to know the value of the pixel in the image, we also want to know the value of the spatial derivatives for that pixel), and the system works at 30 FPS, with a resolution of 640x480 pixels, then, for the first level of the pyramid we need a bandwidth of 0.295 Gbit/s. For

the next levels we need to save a quarter of the previous level. For the 4 levels of the pyramid, we need a total bandwidth of 0.392 Gbit/s which, a DDR2 memory with an 16 data width working at 400 Mhz, resulting in a bandwidth of 12.8 Gbit/s, is more then able to handle. However, we also have to read from the memory to compute the optical flow. For each pixel, in whatever level we are, we need to read 2 times a window of 11 by 11 pixels of the memory. If we do the math, we come to the conclusion that we need to read from the memory at a rate of 94.8 Gbit/s, which is way over the bandwidth available. The solution for this problem is to implement an interface between the memory and the user application that implements a cache memory, to reduce the number of accesses to the memory. This cache can hold the information about the region of the image that is being used to compute the optical flow. The region must be large enough so the optical flow can be computed for multiple pixels before it is necessary to access the memory again. The size of the region also has to take into consideration the resources available in the FPGA, where the algorithm is being implemented. Moreover, we cannot forget that we need more bandwidth available for saving the result of the optical flow, and to send out the final result of the computation.

5.3 Pyramid construction

The construction of the pyramid can be performed in pipeline. As the pixels of the image J, the current video frame, arrive they enter a subsystem where the image is convoluted with tree kernels at the same time, computing the smoothed image as well as the x derivative and the y derivative. Then, the smoothed image is passed for the subsampling stage where the frequency of the pixel clock is reduced, in average, by half. In this stage, half the columns and half the rows are discarded. All this process is then repeated as many times as necessary. In the figure 5.2, the pyramid has 3 levels; however, the best solution is to use 4 levels, and then the system decides how many levels to use in function of the pixel velocity. This way the system will be able to work at higher velocities when needed. In each level of the pyramid, it is saved to the memory the spatial derivatives and the Gaussian smoothed image.

In the next section, it will be discussed how to implement the convolution to compute the smoothing of the image as well as the spatial derivatives.

5.4 2D Convolution

In order to compute the spatial derivatives of the image as well as to save resources and have a lower delay, it is possible to make use of the fact that the kernels are of size 5 by 5 for the Gaussian Kernel, 5 by 1 for the y derivative kernel and 1 by 5 for the x kernel. Using just the block that computes the smoothed image, figure 5.3, we can use the fact that the pixels for computing the spatial derivatives are available and compute them at the same time.

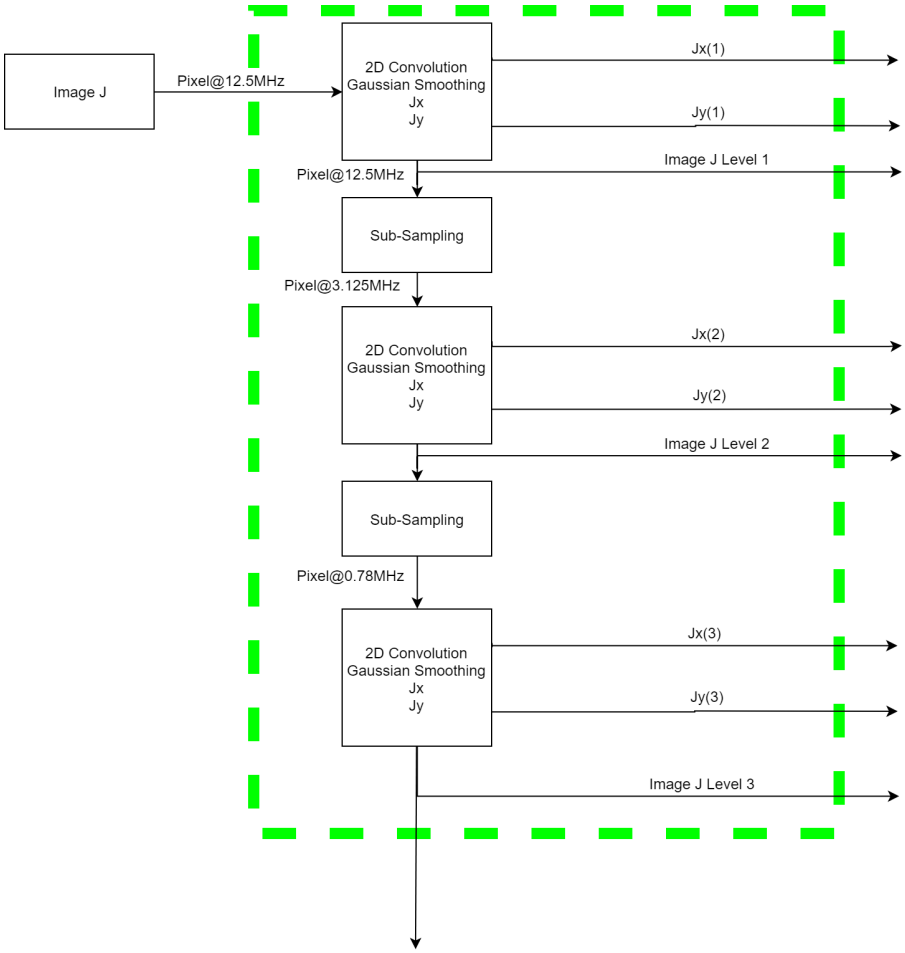


Figure 5.2: Pyramid construction

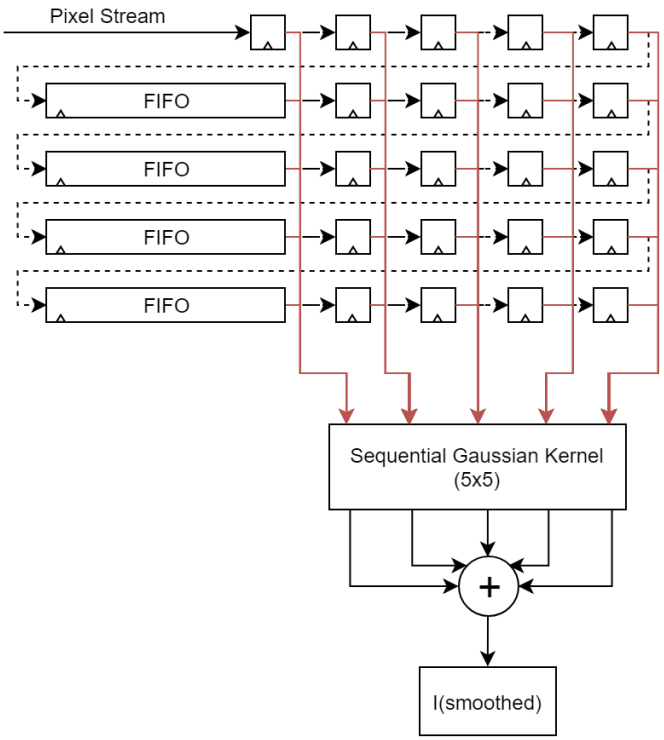
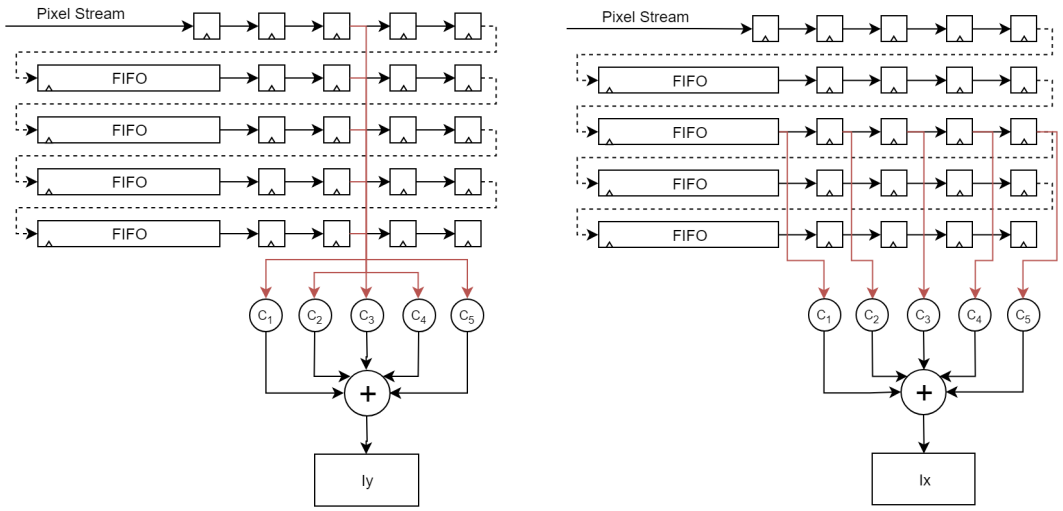


Figure 5.3: Convolution block for computing the smoothed image.

To obtain the window 5 by 5 to compute the smoothed image and the spatial derivatives, we can use the architecture presented in the figure 5.3. The pixels go through 5 registers, that store 1 pixel with 8 bits, in line, that work like a sift register, and then are followed by a FIFO, with size $640-4\times 8$bit, that stores the rest of the image line. This is repeated 4 more times to create the 5 by 5 moving window in the figure. To save resources, we can use the output of the FIFO for computing the window. For computing the smoothed image, a sequential block computes one output pixel in 6 states: 5 states to multiply and add one 5-pixel line by its weights, and an additional state to add the 5 partial sums. This is clocked by a 100 MHz clock, thus allowing 8 clock cycles between the arrival of two image pixels. In the end, the sum of all the values has to be divided by 256, which can be performed by simply discarding the 8 least significant bits. In order to prevent overflow, the multiplication and sum must be performed with unsigned 16 bits.



(a) Schematic for computing the y derivative.

(b) Schematic for computing the x derivative.

Figure 5.4: Spatial derivatives schematic. C_i is the i coefficient of the spatial derivatives.

Using the same FIFOs and registers for the Gaussian kernel, we can compute spatial derivative in the y and x direction, by accessing only to the middle column, figure 5.4a and 5.4b . Also, as the middle value is being multiplied by 0 it can be discarded. Two of the coefficients are 1 and -1, then there is no need for a multiplier, and the other coefficients are 8 and -8, which can be implemented with a a shift of 3 bits to the left. In the end, we need to do a division by 12 (see the section about derivative kernels 3.1.2). To save resources, this is also computed by a sequential block, requiring 6 clock cycles.

5.5 Matrix creation

Assuming we can obtain the values for $I_t(L,k)$, $I_x(L,k)$ as well as $I_y(L,k)$, temporal derivative, x derivative and y derivate, for the current level, L, and iteration, k, of the algorithm, respectively. We can compute the matrix as in the subsection 3.1.4. Observing the figure 5.5, it is required

5 multipliers, 5 adders and 5 accumulators to implement the matrix constructor subsystem in an iterative form. Diaz et al., in [24] uses this stage with 18 bits, although this must be optimized for the hardware architecture where the algorithm is being implemented.

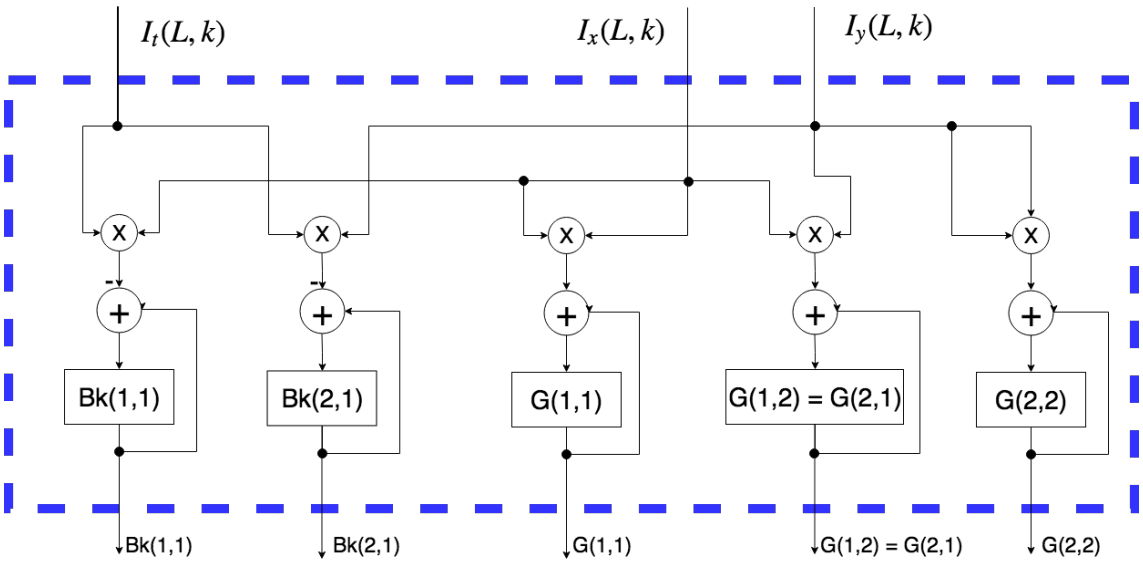


Figure 5.5: Subsystem for matrix creation.

5.6 Matrix solver

In the matrix solver, we solve the least square problem as described in the subsection 3.1.5. We need to compute the determiner of the matrix G, and also the norm of the gradient, in parallel. In the figure 5.6 is a concept of the system to compute the optical flow. It is estimated that we need 7 adders, 10 multipliers and 3 dividers. As it was explained before, in the subsection 3.1.5, we have to decide which value computed, with the two different methods, to use, or if none of the methods was able to compute the optical flow for that pixel. To choose what value to use at the end, a multiplexer chooses which value to use in function of the determinant and the norm. Diaz et al. in [24], presents a study where it was found that the best representation, for maximum clock frequency without much degradation of the optical flow result and higher resource usage, was floating point with 11 bits of mantissa and 7 exponential. Another representation presented in the same study was fixed point with 36 bits, and although the resource usage dropped and the error was reduced, the maximum clock frequency dropped.

5.7 Resource usage estimation

The necessary resources were estimated as follows:

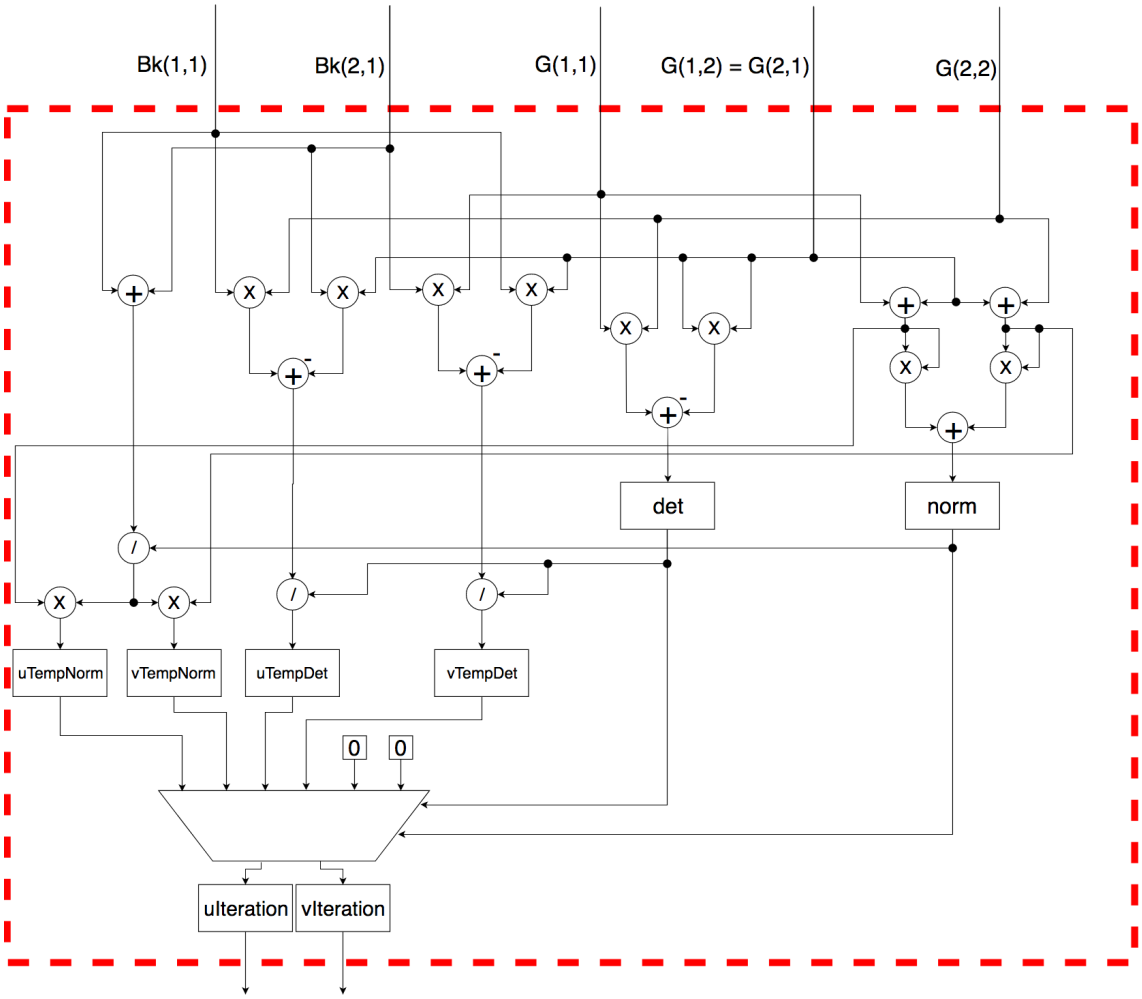


Figure 5.6: Architecture proposal for the solving the system.

- **Adders/Subtractors**: 7 for matrix solver; 5 for the matrix constructor; 24 for the Gaussian smoothing kernel; 8 for the spatial derivatives; 1 for the temporal derivative; 2 for the image warping process. In total, it is estimated to be necessary, at least, 47 adders/subtractors.

- **Multipliers**: 10 for the matrix solver; 5 for the matrix creation; 20 for the Gaussian smoothing kernel. Giving a estimation of the total multipliers needed as 35.

- **Dividers**: 3 for the matrix solver; 8 for the spatial derivatives. In total, it is estimated to be necessary 11 dividers.

- **Memory**: As the encapsulation of the images with the derivatives can be done in a 32bit word, then we have to store 640x480x32bit for the first level of the pyramid, 320x240x32bit for the second, 160x120x32bit for the third and 70x60x32bit for the fourth level. Additionally, we have to store at least 3 images in the RAM. Also, we need to store the optical flow value, u and v, for each pixel in the original frame. Then, it is estimated that we need at least 4.9 MByte for the 3 images and spatial derivatives, and 4.92 MByte to save 2 optical

flow fields, one for the present computation and another for the previous one that is being sent out.

Chapter 6

Conclusions and future work

During this work, the problem of how to estimate the velocity of the AUVs was addressed with special focus on using a vison-based system. It were studied 3 different algorithms with different optimizations and parameters to study the best way to compute the motion field using a monocular camera. It were also discussed the inherent problems to the vision based systems for motion estimation, namely the aperture problem and the parallax effect. Moreover, it was purposed an algorithm to compute the angular and translational velocity of a vehicle using the motion field. It was also addressed the limitations of the algorithm as the maximum velocity and the size of radius of an angular displacement.

Based on test in the chapter 3, it was concluded that the motion field estimation methods more suitable to use were the iterative and pyramidal Lucas and Kanade and the block matching, as it were the only methods that could cope with the large displacements between frames. In the chapter 4, is is described the tests performed to validate the velocity estimation algorithm using, a Matlab generated sequence, for the angular velocity, and a vehicle accelerating to a known velocity, for the translational velocity. The results show that the algorithm performed as expected. It was also performed tests to evaluate the results, for the velocity estimation, using the two different algorithms for the motion field estimation. We conclude that the algorithm that should be used is the iterative and pyramidal Lucas and Kanade, which showed to be more robust in difficult conditions, and also has a better accuracy over the block matching. It was also determined the parameters that should be used to yield the best results and keeping the algorithm simple enough to implement in hardware. The relative errors for the translational velocity and angular velocity, using the iterative and pyramidal Lucas and Kanade method, in the tests performed, were both below 3%.

Finally, it was presented a hardware architecture to implement the iterative and pyramidal Lucas and Kanade. It was also presented an estimation of the resource usage for an FPGA implementation of the motion field estimation core. Moreover, it was analysed details of the implementation, such as the number representation, in binary, and memory usage and bandwidth. A scheme for the pyramidal construction, and spatial derivative construction, using 2D convolutions is also presented.

Overall, throughout this work, it were presented and studied different optical flow algorithms, that were implemented, in a simulation environment (Matlab), and were evaluated their performances and defined their different parameters, in order to optimize our final solution, which allowed to obtain an accurate velocity estimation. Moreover, it was also presented an algorithm for angular and translational velocity estimation, also implemented in a simulation environment, that was able to produce accurate results, when used in conjunction with the optical flow algorithms. Every stage of the algorithms is also detailed in this document for an easy result replication and implementation in different platforms.

Finally, it is presented an estimation of the resource consumption and an hardware architecture is proposed feasible to be implemented in a FPGA. The work developed in this dissertation is a starting point and an important step for the implementation in hardware and in an embedded system, but more work needs to be done to fully implement a final system.

6.0.1 Future work

As a continuation of this work, we suggest the implementation of the core, for the motion estimation algorithm, iterative and pyramidal Lucas and Kanade, following the hardware architecture presented in this dissertation. Also, it is suggested the implementation of the velocity estimation algorithm, that will use the results of the motion estimation core, to compute the velocity of the vehicle. At the end, the integration of the both systems with a VGA camera and a set of tests should be performed to characterize the final solution.

Due to parallax effect, it is necessary to determine the distance to the object that we are computing the velocity relative to. In order to obtain this distance, we believe it is essential the development of a distance measurement system based on computer vision. Indeed, our suggestion consists in using the video, already recorded for the velocity measurement, to measure this distance. This can be further improved by measuring the distance to the object for the different frame regions, as previously described. Hence, it would enable us to correct the parallax effect when inside the frame are regions that are closer and regions that are further way.

Although this work focus on the velocity estimation of an AUV, the motion field computed through the use of the iterative and pyramidal Lucas and Kanade can also be used for other applications inside the robotic field, such as movement segmentation, security systems or motion detection. Then, the system can be further improved by creating a framework, for the computation of the Lucas and Kanade, so it can be used in parallel for others applications. Other type of vehicles where the camera is facing a plantar structure, such as as quadcopters or other unmanned aerial vehicles can also benefit from the implementation of this algorithm.

Further development and improvements through the integration of this system with other systems should be done to compute the angular velocity, since it is difficult, based purely in vision, to know when the vehicle is rotating if the translational velocity is much higher than the angular velocity.

References

[1] L. Paull, S. Saeedi, M. Seto, and H. Li. Auv navigation and localization: A review. *IEEE Journal of Oceanic Engineering*, 39(1):131–149, 2014. doi:10.1109/JOE.2013.2278891.

[2] J. Corso. Motion and optical flow, 2014. URL: https://courses.engr.illinois.edu/cs543/sp2017/.

[3] A. Shukla and H. Karki. Application of robotics in offshore oil and gas industry— a review part ii. *Robotics and Autonomous Systems*, 75:508–524, 2016. doi:https://doi.org/10.1016/j.robot.2015.09.013.

[4] C. Mai, S. Pedersen, L. Hansen, K. L. Jepsen, and Y. Zhenyu. Subsea infrastructure inspection: A review study. In *2016 IEEE International Conference on Underwater System Technology: Theory and Applications (USYS)*, pages 71–76. doi:10.1109/USYS.2016.7893928.

[5] J. J. Leonard and A. Bahr. *Autonomous Underwater Vehicle Navigation*, pages 341–358. Springer International Publishing, Cham, 2016. doi:10.1007/978-3-319-16649-0_14.

[6] S. Baker, D. Scharstein, J. P. Lewis, S. Roth, M. J. Black, and R. Szeliski. A database and evaluation methodology for optical flow. *International Journal of Computer Vision*, 92(1):1–31, 2010. doi:10.1007/s11263-010-0390-2.

[7] P. A. Work, K. A. Haas, Z. Defne, and T. Gay. Tidal stream energy site assessment via three-dimensional model and measurements. *Applied Energy*, 102:510–519, 2013. doi:10.1016/j.apenergy.2012.08.040.

[8] J. Jung, Y. Lee, D. Kim, D. Lee, H. Myung, and H. T. Choi. Auv slam using forward/downward looking cameras and artificial landmarks. In *2017 IEEE Underwater Technology (UT)*, pages 1–3. doi:10.1109/UT.2017.7890307.

[9] L. Pan-Mook, J. Bong-Hwan, and L. Chong-Moo. A docking and control system for an autonomous underwater vehicle. In *OCEANS '02 MTS/IEEE*, volume 3, pages 1609–1614 vol.3. doi:10.1109/OCEANS.2002.1191875.

[10] F. Yang and A. Balasuriya. Target tracking by underwater robots. In *2001 IEEE International Conference on Systems, Man and Cybernetics. e-Systems and e-Man for Cybernetics in Cyberspace (Cat.No.01CH37236)*, volume 2, pages 696–701 vol.2. doi:10.1109/ICSMC.2001.972995.

[11] B. Horn. *Robot Vision*. MIT press, 1986.

[12] B. K. P. Horn and B. G. Schunck. Determining optical flow. *Artificial Intelligence*, 17(1):185–203, 1981. doi:https://doi.org/10.1016/0004-3702(81)90024-2.

[13] B. D. Lucas and T. Kanade. An iterative image registration technique with an application to stereo vision. 1981.

[14] M. J. Black and P. Anandan. A framework for the robust estimation of optical flow. In *1993 (4th) International Conference on Computer Vision*, pages 231–236. doi:10.1109/ICCV.1993.378214.

[15] T. Amiaz, E. Lubetzky, and N. Kiryati. Coarse to over-fine optical flow estimation. *Pattern recognition*, 40(9):2496–2503, 2007.

[16] A. M. Pinto, A. P. Moreira, M. V. Correia, and P. G. Costa. A flow-based motion perception technique for an autonomous robot system. *Journal of Intelligent and Robotic Systems*, 75(3-4):475–492, 2013. doi:10.1007/s10846-013-9999-z.

[17] S. Baker, D. Scharstein, J. P. Lewis, S. Roth, M. J. Black, and R. Szeliski. Optical flow evaluation results. URL: http://vision.middlebury.edu/flow/eval/results/results-e1.php.

[18] J. L. Barron, D. J. Fleet, and S. S. Beauchemin. Performance of optical flow techniques. *International Journal of Computer Vision*, 12(1):43–77, 1994. doi:10.1007/BF01420984.

[19] F. Barranco, M. Tomasi, J. Diaz, M. Vanegas, and E. Ros. Parallel architecture for hierarchical optical flow estimation based on fpga. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 20(6):1058–1067, 2012. doi:10.1109/TVLSI.2011.2145423.

[20] M. Komorkiewicz, T. Kryjak, and M. Gorgon. Efficient hardware implementation of the horn-schunck algorithm for high-resolution real-time dense optical flow sensor. *Sensors*, 14(2):2860, 2014.

[21] M. Kunz, A. Ostrowski, and P. Zipf. An fpga-optimized architecture of horn and schunck optical flow algorithm for real-time applications. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. doi:10.1109/FPL.2014.6927406.

[22] Z. Wei, D.-J. Lee, and B. E. Nelson. Fpga-based real-time optical flow algorithm design and implementation. *Journal of Multimedia*, 2(5), 2007.

[23] J. Diaz, E. Ros, F. Pelayo, E. M. Ortigosa, and S. Mota. Fpga-based real-time optical-flow system. *IEEE Transactions on Circuits and Systems for Video Technology*, 16(2):274–279, 2006. doi:10.1109/TCSVT.2005.861947.

[24] J. Díaz, E. Ros, R. Agís, and J. L. Bernier. Superpipelined high-performance optical-flow computation architecture. *Computer Vision and Image Understanding*, 112(3):262–273, 2008. doi:https://doi.org/10.1016/j.cviu.2008.05.006.

[25] V. Mahalingam, K. Bhattacharya, N. Ranganathan, H. Chakravarthula, R. R. Murphy, and K. S. Pratt. A vlsi architecture and algorithm for lucas and kanade-based optical flow computation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 18(1):29–38, 2010. doi:10.1109/TVLSI.2008.2006900.

[26] H. S. Seong, C. E. Rhee, and H. J. Lee. A novel hardware architecture of the lucas and kanade optical flow for reduced frame memory access. *IEEE Transactions on Circuits and Systems for Video Technology*, 26(6):1187–1199, 2016. doi:10.1109/TCSVT.2015.2437077.

[27] J. L. Martín, A. Zuloaga, C. Cuadrado, J. Lázaro, and U. Bidarte. Hardware implementation of optical flow constraint equation using fpgas. *Computer Vision and Image Understanding*, 98(3):462–490, 2005. doi:https://doi.org/10.1016/j.cviu.2004.10.002.

[28] R. Rustam, N. H. Hamid, and F. A. Hussin. Fpga-based hardware implementation of optical flow constraint equation of horn and schunck. In *2012 4th International Conference on Intelligent and Advanced Systems (ICIAS2012)*, volume 2, pages 790–794. doi:10.1109/ICIAS.2012.6306121.

[29] A. M. G. Pinto. *Visual Motion Analysis Based on a Robotic Moving System*. Thesis, 2014. doi:http://hdl.handle.net/10216/73552.

[30] L. R. G. Carrillo, I. Fantoni, E. Rondon, and A. Dzul. Three-dimensional position and velocity regulation of a quad-rotorcraft using optical flow. *IEEE Transactions on Aerospace and Electronic Systems*, 51(1):358–371, 2015.

[31] R. A. Haddad and A. N. Akansu. A class of fast gaussian binomial filters for speech and image processing. *IEEE Transactions on Signal Processing*, 39(3):723–727, 1991. doi:10.1109/78.80892.

[32] J. W. Brandt. Improved accuracy in gradient-based optical flow estimation. *International Journal of Computer Vision*, 25(1):5–22, 1997. doi:10.1023/A:1007987001439.

[33] J.-Y. Bouguet. Pyramidal implementation of the affine lucas kanade feature tracker description of the algorithm. *Intel Corporation*, 5(1-10):4, 2001.

[34] XILINX. User guide: Spartan-6 fpga memory controller, 2010. URL: https://www.xilinx.com/support/documentation/user_guides/ug388.pdf.

[35] XILINX. User guide: Memory interface solutions, 2010. URL: https://www.xilinx.com/support/documentation/ip_documentation/ug086.pdf.